Grzegorz Gałęzowski

Test-Driven Development

Extensive Tutorial

Test-Driven Development: Rozbudowany samouczek

Grzegorz Gałęzowski i Borysław Bobulski

Ta książka jest do kupienia na http://leanpub.com/web-of-objects

Wersja opublikowana 2019-02-27

Spis treści

Przedmowa . . . 1

Dedykacja . . . 2

Podziękowania! . . . 3

O przykładach kodu . . . 4
- Uwagi dla programistów języka C# . . . 4
- Uwagi dla programistów języka Java . . . 4

Część 1: Same podstawy . . . 8

Motywacja – pierwszy krok w uczeniu się TDD . . . 9
- Jakie jest TDD? . . . 10
- Zaczynajmy ! . . . 12

Niezbędne narzędzia . . . 13
- Test framework . . . 13
- Framework do mockowania . . . 20
- Generator wartości anonimizowanych . . . 27
- Podsumowanie . . . 29

To nie (tylko) test . . . 31
- Kiedy test staje się czymś więcej . . . 31
- Testy w świecie programistów . . . 32
- Raczej specyfikacja niż zbiór testów . . . 33
- Różnice między “wykonywalnymi” specyfikacjami i tymi “tradycyjnymi” . . . 34

Programowanie poprzedzone wymaganiem (Wymaganie-najpierw) . . . 36
- Po co pisać specyfikację po fakcie? . . . 36
- “Najpierw-test” oznacza patrzenie na niepowodzenie . . . 38
- “Test-Po” często kończy jako “Test-Nigdy” . . . 43
- “Test-Po” często prowadzi do ponownego projektowania . . . 44
- Podsumowanie . . . 45

Poćwiczmy to, czego się właśnie nauczyliśmy **46**
Pozwól mi opowiedzieć sobie historię 46
Akt 1: Samochód 47
Akt 2: Tytuł dla Klienta 47
Akt 3: Test-Driven Development 51
Epilog 66

Odnajdźmy się odrobinę **68**

Przedmowa

Dedykacja

Ad Deum qui laetificat iuventutem meam.

Mej ukochanej zonie Monice i naszemu ukochanemu synowi Danielowi.

Podziękowania!

Chciałbym podziękować następującym osobom (wymienionym w kolejności alfabetycznej) za wartościowe uwagi, sugestie, poprawki literówek i tym podobny wkład

- Brad Appleton
- Borysław Bobulski
- Chris Kucharski
- Daniel Dec
- Daniel Żołopa (projekt okładaki)
- Donghyun Lee
- Łukasz Maternia
- Marek Radecki
- Martin Moene
- Michael Whelan
- Polina Kravchenko
- Rafał Bigaj
- Reuven Yagel
- Rémi Goyard
- Robert Pająk
- Wiktor Żołnowski

Książka ta nie jest niczym, czego by dotychczas nie napisano. Prezentuje zagadnienia, które już poruszono a które ja, kolejny raz, podjąłem.
Dlatego też, chciałbym bardzo podziękować moim mentorom i autorytetom, zajmujących się test-driven development i projektowaniem zorientowanym obiektowo, od których zdobyłem większość swojej wiedzy (wymienieni niżej w kolejności alfabetycznej)

- Amir Kolsky
- Dan North
- Emily Bache
- Ken Pugh
- Kent Beck
- Mark Seemann
- Martin Fowler
- Nat Pryce
- Philip Schwarz
- Robert C. Martin
- Scott Bain
- Steve Freeman

O przykładach kodu

Uwagi dla programistów języka C#

Językiem, który wybrano dla przykładów jest C#, aczkolwiek zrobiłem kilka wyjątków od typowej C# konwencji.

Usunięcie "I" z nazw interfejsów

Osobiście, nie jestem fanem używania `ISomething` dla nazw interfejsu, więc zdecydowałem się nie umieszczać przedrostka `I` nawet jeśli większość programistów C# by tego oczekiwała. Mam nadzieję, że tym razem mi wybaczą.

Konstrukcje języka charakterystyczne dla C#

Większość kodu w tej książce to nie jest typowy C# ze swoimi idiomami. Próbowałem unikać właściwości (properties), zdarzeń (events) i większości nowoczesnych funkcji. Moim celem jest umożliwienie użytkownikom innych języków (zwłaszcza Javy) korzystanie z tej książki.

Używanie podkreśleń w nazwach pól

Niektórzy to lubią, inni nie. Postanowiłem trzymać konwencję umieszczania znaku podkreślenia (_) przed nazwą pola klasy.

Uwagi dla programistów języka Java

Językiem, w którym ukazano przykłady kodu, jest C#. Zaznaczyłem jednak, że chcę, aby książka była jak najmniej związana z konkretną technologią, by umożliwić programistom języka Java czerpanie z niej korzyści. Próbowałem używać minimalnej liczby funkcji specyficznych dla C#, a w kilku miejscach nawet robiłem uwagi skierowane do programistów Java, aby im ułatwić zrozumienie. Jest jednak kilka rzeczy, których nie mogłem uniknąć, toteż stworzyłem listę opisującą kilka różnic między Javą i C#, co może się przydać.

Konwencja nazewnictwa

Większość języków ma swoje domyślne konwencje nazewnictwa. Na przykład w języku Java nazwa klasy jest zapisana za pomocą PascalCase (np. `UserAccount`), metody i pola są zapisywane za pomocą camelCase, np. "payTaxes" a stałe/pola tylko do odczytu są zapisywane wielkimi literami z podkreśleniem (np. `CONNECTED_NODES`).

C# używa pascalCase dla nazw klas i metod (np. `UserAccount`, `PayTaxes`, `ConnectedNodes`). W przypadku pól istnieje kilka konwencji nazewnictwa. Wybrałem tą, która zaczyna się od znaku podkreślenia (np. `_myDependency`). Są jeszcze inne drobne różnice, ale z tymi się spotkasz najczęściej.

słowo kluczowe `var`

By uczynić zapis bardziej zwięzłym, zdecydowałem się użyć w przykładach słowa kluczowego `var`. To słowo kluczowe służy do automatycznego wnioskowania jakiego typu jest zmienna, np.

```
1  var x = 123; // wnioskuję, że x jest liczbą całkowitą
```

Oczywiście, to nie jest typowanie dynamiczne (dynamic typing) - wszystko jest określane na etapie kompilacji.

Jeszcze jedno - słowo kluczowe "var" może być użyte tylko wtedy, kiedy można wywnioskować z jakim typem mamy do czynienia, w nnych wypadkach musiałem deklarować typy jawnie, jak w przypadku:

```
1  List<string> list = null; //nie można użyć var by wnioskować jakiego typu to lista
```

słowo kluczowe `string`

C# ma typ `String`, podobnie jak Java. C# pozwala jednak na wpisanie nazwy tego typu jako słowa kluczowego, np. `string` zamiast `String`. To jest tylko lukier składniowy (syntactic sugar), który jest domyślnie używany przez społeczność C#.

Atrybuty zamiast adnotacji

W języku C# istnieją atrybuty. Są używane w tym samym celu, co adnotacje (anntoations) w Javie. Tak więc, gdy widzisz:

```
1  [Whatever]
2  public void doSomething()
```

myśl:

```
@Whatever
public void doSomething()
```

`readonly` i `const` w miejscu `final`

Tam, gdzie w Javie używa słowa `final` dla stałych (wraz z `static`) i pól tylko do odczytu, w C# używa się dwóch słów kluczowych: `const` i `readonly`. Bez wchodzenia w szczegóły, za każdym razem, gdy zobaczysz coś takiego:

```
public class User
{
  // a constant with literal:
  private const int DefaultAge = 15;

  // a "constant" object:
  private static readonly TimeSpan DefaultSessionTime
    = TimeSpan.FromDays(2);

  // a read-only instance field:
  private readonly List<int> _marks = new List<int>();
}
```

myśl:

```
public class User {
  //a constant with literal:
  private static final int DEFAULT_AGE = 15;

  //a "constant" object:
  private static final Duration
    DEFAULT_SESSION_TIME = Duration.ofDays(2);

  // a read-only instance field:
  private final List<Integer> marks = new ArrayList<>();
}
```

Konstrukcja `List<T>`

Jeśli jesteś programistą języka Java to zauważ, że w C# `List<T>` nie jest interfejsem, ale konkretną klasą. Ten typ jest zwykle używany tam, gdzie Ty używałbyś `ArrayList`

Typy uogólnione (generyczne)

Jedną z największych różnic między Javą i C # jest to, jak traktuje się typy generyczne. Po pierwsze, C# pozwala na używanie typów prostych (prymitywnych) w deklaracjach typów uogólnionych, więc możesz napisać `List<int>` w języku C#, podczas gdy w Javie musisz napisać `List<Integer>`.

Inną różnicą jest to, że w języku C# nie ma wymazywania typów, tak jak w Javie. Kod napisany w C# zachowuje wszystkie informacje o typie generycznym w czasie wykonywania. To istotnie wpływa na to, jak są projektowane i jak można używać API korzystającego z typów generycznych.

Definicja klas generycznych i ich tworzenie, w Javie i C# wyglądają mniej więcej tak samo. Istnieje jednak różnica na poziomie metody. Metoda generyczna w Javie wygląda tak:

```
1  public <T> List<T> createArrayOf(Class<T> type) {
2      ...
3  }
```

a używamy jej tak:

```
1  List<Integer> ints = createArrayOf(Integer.class);
```

podczas, gdy w C# ta sama metoda będzie zdefiniowana tak:

```
1  public List<T> CreateArrayOf<T>()
2  {
3      ...
4  }
```

i używana w taki sposób:

```
1  List<int> ints = CreateArrayOf<int>();
```

Różnice te są widoczne w projekcie biblioteki, używanej w tej książce do generowania danych testowych[^anylibraryauthor]. W wersji C#, generujemy dane testowe, pisząc:

```
1  var data = Any.Instance<MyData>();
```

w jezyku Java zaś:

```
1  MyData data = Any.instanceOf(MyData.class);
```

Część 1: Same podstawy

W tej części przedstawiam podstawy filozofii TDD (test-driven development - sterowane testami wytwarzanie oprogramowania) i sposoby pisania kodu, bez zbytniego wchodzenia w zaawansowane aspekty takie jak wprowadzanie TDD do systemów zorientowanych obiektowo, gdzie współpracuje ze sobą mnóstwo obiektów (piszę o tym w części drugiej). Większość przykładów, przytoczonych w tej części, dotyczy zaprojektowania pojedynczego obiektu.

Zanim przejdę do konkretnych zastosowań TDD, skoncentruje się nad samą istotą tego podejścia. Potem - powoli - wprowadzę konkretne pojęcia w łatwy do zrozumienia sposób.

Po przeczytaniu części pierwszej, dzięki TDD, będziesz w stanie całkiem sprawnie projektować klasy, które nie zależą od innych klas (ani nie zależą od jakichkolwiek zasobów systemowych).

Motywacja – pierwszy krok w uczeniu się TDD

Piszę tę książkę, ponieważ jestem entuzjastą TDD. Wierzę, że TDD ma przewagę nad innymi sposobami wytwarzania oprogramowania, których używałem. Myślę, że wielu programistów podziela moje przekonania. To skłania mnie do zadania pytania - dlaczego więcej osób nie miałoby uczyć się TDD, dlaczego nie miałoby stosować TDD w swojej pracy? Wciąż nie mogę stwierdzić, że TDD jest głównym nurtem w wytwarzaniu oprogramowania. Podczas mojej kariery zawodowej nie widziałem wystarczająco dużo przykładów, które by potwierdzały taki stan rzeczy.

Drogi czytelniku! Już zdobyłeś mój szacunek, bo zdecydowałeś się sięgnąć po książkę, zamiast budować swoje rozumienie TDD na podstawie miejskich legend i swoich wyobrażeń. Nieważne, czy to Twoja pierwsza książka podejmująca temat TDD, czy też miałeś styczność z innymi - jestem zaszczycony i szczęśliwy, że dziś wybrałeś moją książkę. Mam nadzieję, że przeczytasz ją od deski do deski. Gdybyś jeszcze się wahał, chcę zadać Ci pomocnicze pytanie, które pomoże Ci stwierdzić, czy naprawdę masz ochotę na czytanie. Dlaczego właściwie chcesz się uczyć o TDD?

Kwestionując Twoją motywację, nie staram się zniechęcać Cię do czytania. Raczej chciałbym, byś dobrze wiedział co chcesz osiągnąć przeczytawszy moją książkę. Kilka lat temu uczyłem pewną osobę, która zainteresowała się TDD. Zaczęliśmy razem pracować nad małym projektem, by ta osoba mogła nabyć niezbędnych umiejętności poprzez praktykę, ja zaś siedziałem obok niej, udzielając wskazówek. Tych lekcji było trzy, lub cztery - potem mój uczeń zrezygnował mając "pilniejsze rzeczy do zrobienia" i "nie mając czasu". Od tamtej pory, nie wykorzystywał TDD w pracy ani nie starał się zrozumieć niczego więcej. Nawet dzisiaj, zastanawiam się, co było jego motywacją i dlaczego ta wyparowała?

Innym powodem dla którego można chcieć uczyć się TDD, są błędne oczekiwania. Niektórzy z nas mają mgliste wyobrażenie o rzeczywistych kosztach i zyskach, jakie daje TDD. Wiedząc, że jest cenione i chwalone przez innych, można wyciągnąć wnioski, że to będzie idealne rozwiązanie również dla nas. Dla przykładu, ktoś oczekujący poprawnie działającego kodu mógł usłyszeć, że dzięki TDD "kod staje się bardziej przetestowany". Jeśli nie mamy wiedzy, dlaczego warto wprowadzać TDD do projektu, możemy uważać, iż należy pisać najpierw testy przed kodem, po to tylko, by zapewnić 100% pokrycia kodu. Nie zrozumcie mnie źle - to częściowo może być prawdą, aczkolwiek w takim podejściu gubimy istotę TDD. Co więcej, gdy od TDD oczekiwaliśmy czegoś, czego ono nie daje, możemy być bardzo rozczarowani. Słyszałem wiele osób, które twierdziły "nie potrzebuję TDD, bo muszę mieć testy systemowe dające większą pewność, co do poprawnego działania produktu" albo "po co mi testy jednostkowe[1] kiedy już mam testy integracyjne, smoke-testy, sanity-testy, exploration-testy, etc…?". Wiele razy byłem świadkiem, że porzucano idee TDD zanim ją w ogóle zrozumiano.

[1] Nawiasem mówiąc, TDD to nie tylko testy jednostkowe, jeszcze do tego dojdziemy.

Czy nauka TDD ma dla Ciebie priorytet? Czy jesteś zdeterminowany, by wypróbować TDD i naprawdę się tego nauczyć? Jeśli jest inaczej - hej - słyszałem, że nowy sezon "Gry o Tron" pojawił się w telewizji, czemu nie miałbyś zająć się właśnie nim? Dobra, tylko się przekomarzam. Mówi się, że zasady TDD są łatwe do zrozumienia, ale ciężko być prawdziwym ekspertem ("easy to learn, hard to master"[2]). Dlatego, bez odrobiny odwagi, będzie ciężko. Szczególnie, że zamierzam wprowadzać Cię w temat powoli, stopniowo, byś otrzymał lepsze wyjaśnienie różnych technik i sposobów pisania.

Jakie jest TDD?

Razem z bratem lubiliśmy grać w gry video gdy byliśmy dziećmi – szczególnie miło wspominam grę Tekken 3 – japoński beat'em up na Sony Playstation (PSX). Ukończenie gry wszystkimi zawodnikami i odblokowanie wszystkich ukrytych bonusów, mini-gier etc. zajmowało jeden dzień. Ktoś mógłby powiedzieć, że od tego momentu gra nie oferuje niczego więcej. Dlaczego więc, z bratem, graliśmy w nią ponad rok?

[2]Nie wiem, kto pierwszy to powiedział, przeszukałem Internet i znalazłem ten zwrot w kilku miejscach, gdzie żaden z piszących nie informował, kogo cytuje - więc postanowiłem tylko wspomnieć, że nie jestem autorem tych słów.

Tekken3

To dlatego, że każdy wojownik w grze posiadał mnóstwo kombosów, kopnięć i uderzeń rękami, które można było łączyć na różne sposoby. Niektóre dało się zastosować tylko w określonych sytuacjach, inne mogłem użyć prawie zawsze bez ryzyka narażenia się na kontratakt. Mogłem zejść z lini ataku przeciwnika, a także byłem w stanie wykopać przeciwnika w powietrze, gdzie nie mógł już blokować moich ciosów, a potem wykonać dodatkowy atak zanim upadł na ziemię. Ta powietrzna technika nazywa się "juggles". W tamtych czasach pojawiały się czasopisma, które każdego miesiąca publikowały listę nowo odkrytych "juggles", nie pozwalając - w ten spoób - zgasnąć fascynacji graczy przez ponad rok.

Tak, łatwo było się nauczyć grać w Tekken – mogłem poświęcić zaledwie jedną godzinę trenując najważniejsze ruchy postaci i byłem w stanie już "używać" danego zawodnika, ale wiedziałem, że lepiej bym walczył gdybym zdobył doświadczenie i wiedzę, które techniki są ryzykowne, a które nie, których ataków używać w określonych sytuacjach, jak łączyć je ze sobą, jak zmniejszyć możliwość kontrataku. Nic dziwnego, że wkrótce pojawiło się wiele turniejów, gdzie gracze mogli walczyć o chwałę, sławę i nagrody. Nawet dziś, można obejrzeć niektóre z tych legendarnych pojedynków na YouTube.

TDD jest jak Tekken. Prawdopodobnie słyszałeś pojęcie "red-green-refactor" lub ogólną zasadę

“napisz najpierw test, potem kod”, może nawet przeprowadziłeś eksperyment, gdzie próbowałeś zaimplementować sortowanie bąbelkowe lub jakąś inną rzecz zaczynając od napisania testu. To wszystko jest jak uczenie się Tekkena przez sprawdzanie każdego ataku na przeciwniku, który nie może się ruszać, bez całego kontekstu realnej gry, który czyni pojedynek naprawdę wymagającym. Chociaż uważam takie ćwiczenia “na sucho” za bardzo użyteczne (sam zrobiłem dużo takich), to największe korzyści daje zrozumienie, jak TDD jest używane w prawdziwym życiu.

Niktórzy ludzie z którymi rozmawiam o TDD określają to, co do nich mówię, jako naprawdę demotywujące – “jest tak wiele rzeczy, na które muszę uważać, że nigdy nie chcę zaczynać!” Luz, nie panikuj – przypomnij sobie, jak po raz pierwszy próbowałeś jeździć na rowerze – nie zaprzątałeś sobie głowy tym, że musisz znać jakieś przepisy drogowe i że trzeba przestrzegać znaków drogowych, to Cię wcale nie powstrzymało, prawda?

Uważam, że TDD jest fascynujące i w ogóle sprawia, że pisanie kodu ekscytuje. Niektórzy faceci w moim wieku już myślą, że wiedzą wszystko o kodowaniu, nudzą się z tym i nie mogą się doczekać, aż przejdą do “menadżerki”, wymagań lub analizy biznesowej, ale hej! Oto pojawia się nowy zestaw technik, które sprawią, że moja kariera programisty znów będzie wyzwaniem! Mogę nabyć umiejętności, które będę w stanie zastosować podczas pracy z wieloma różnymi technologiami i językami, a to uczyni mnie lepszym programistą! Czy to nie jest coś, do czego warto dążyć?

Zaczynajmy !

W tym rozdziale próbowałem sprowokować Cię do przemyślenia swojej postawy i motywacji. Jeśli nadal jesteś zdeterminowany by nauczyć się TDD czytając tę książkę (a mam nadzieję, że tak), to zacznijmy pracę!

Niezbędne narzędzia

Czy kiedykolwiek oglądałeś film Karate Kid, starą lub nową wersję? W obu chodzi o to, że kiedy "dzieciak" zaczyna uczyć się od swojego mistrza karate (lub kung-fu), otrzymuje najprostsze zadanie do powtarzania (jak zdjęcie kurtki i założenie jej), nie wiedząc jeszcze po co to robi i gdzie go to zaprowadzi. Albo, włącz sobie pierwszy film Rocky (tak, ten z udziałem Sylvestra Stallone'a), w którym Rocky ściga kurczaka, aby trenować zwinność.

Kiedy po raz pierwszy próbowałem nauczyć się grać na gitarze, znalazłem dwie porady w Internecie: pierwszą było opanowanie pojedynczego, trudnego utworu. Druga rada polegała na graniu na jednej strunie, nauczeniu się jak ta struna może brzmieć i próbie zagrania melodii ze słuchu właśnie na tej jednej strunie. Chyba nie muszę dodawać, że ta druga rada działała lepiej?

Szczerze mówiąc, mógłbym od razu rozpocząć od podstawowych technik TDD, ale wydaje mi się, że byłoby to tak, jakbym postawił Cię na ringu z bardzo wymagającym przeciwnikiem - prawdopodobnie zniechęciłbyś się przed zdobyciem niezbędnych umiejętności. Zamiast wyjaśniać, jak się wygrywa wyścigi, w tym rozdziale przyjrzymy się raczej, jakie błyszczące samochody będziemy prowadzić.

Innymi słowy, zaprezentuję króko trzy narzędzia, z których będziemy korzystać w tej książce

W tym rozdziale upraszczam niektóre rzeczy tylko po to, abyś zaczął działać bez wchodzenia w filozofię TDD (skojarz: lekcje fizyki w szkole podstawowej). Nie przejmuj się :-), nadrobię to w nadchodzących rozdziałach!

Test framework

Pierwszym narzędziem, które wykorzystamy, będzie test framework (framework testujący). *W języku polskim nie ma odpowiednika słowa "framework", najbliżej jest słowo "platforma" lub "struktura". Framework to coś, co wyznacza pewne ramy, definiuje struktury do wykorzystania, dostarcza biblioteki, funkcje - w tym przypadku funkcje pomagające w pisaniu i wykonaniu testów.*

Załóżmy, na potrzeby naszego wprowadzenia, że mamy aplikację, która przyjmuje dwie liczby z linii poleceń, mnoży je i wypisuje wynik na konsoli. Kod jest dość prosty:

```
public static void Main(string[] args)
{
  try
  {
    int firstNumber = Int32.Parse(args[0]);
    int secondNumber = Int32.Parse(args[1]);

    var result =
      new Multiplication(firstNumber, secondNumber).Perform();

    Console.WriteLine("Result is: " + result);
  }
  catch(Exception e)
  {
    Console.WriteLine("Multiplication failed because of: " + e);
  }
}
```

Teraz załóżmy, że chcemy sprawdzić, czy program daje prawidłowe wyniki. Najbardziej oczywistym sposobem sprawdzenia byłoby wywołanie go z linii poleceń - ręcznie - za pomocą kilku przykładowych argumentów, następnie sprawdzenie wyników programu na konsoli i porównanie ich z tym, co czekaliśmy. Taka sesja testowa może wyglądać następująco:

```
C:\MultiplicationApp\MultiplicationApp.exe 3 7
21
C:\MultiplicationApp\
```

Jak widać, nasz program daje wynik 21 dla mnożenia 3 przez 7. Jest to poprawne, więc zakładamy, że program zdał test.

Co się stanie, jeśli program będzie miał również zaimplementowane dodawanie, odejmowanie, dzielenie, całkowanie itp.? Ile razy będziemy musieli go ręcznie wywołać na różne sposoby, by upewnić się, że każda operacja, po naszych zmianach, wciąż działa poprawnie? Czy nie byłoby to czasochłonne? Ale czekaj, jesteśmy programistami, prawda? Tak więc możemy napisać programy do testowania dla nas! Poniżej znajdziesz kod źródłowy innej aplikacji, który używa naszej klasy Multiplication, ale w nieco inny sposób niż robił to nasz wcześniejszy program:

```
public static void Main(string[] args)
{
  var multiplication = new Multiplication(3,7);

  var result = multiplication.Perform();

  if(result != 21)
  {
    throw new Exception("Failed! Expected: 21 but was: " + result);
  }
}
```

Wygląda prosto, prawda? Teraz na tym kodzie oprzemy bardzo prymitywny szkielet testowy - by pokazać fragmenty, z których składają się frameworki testujące. Pierwszym krokiem w tym kierunku będzie wyodrębnienie sprawdzenia wyniku (`result`) do metody, którą będzie można używać wielokrotie. Po tym wszystkim, w mgnieniu oka dodamy do aplikacji dzielenie, pamiętasz? No to jedziemy:

```
public static void Main(string[] args)
{
  var multiplication = new Multiplication(3,7);

  var result = multiplication.Perform();

  AssertTwoIntegersAreEqual(expected: 21, actual: result);
}

// Wyodrębiony kod:
public static void AssertTwoIntegersAreEqual(
  int expected, int actual)
{
  if(actual != expected)
  {
    throw new Exception(
      "Failed! Expected: "
        + expected + " but was: " + actual);
  }
}
```

Zauważ, że nazwę tej wyodrębnionej metody zacząłem od “Assert” - wkrótce wrócimy do nazewnictwa, na razie przyjmijmy, że jest to dobra nazwa dla metody, która sprawdza, czy wynik pasuje do naszych oczekiwań. Ostatnim krokiem będzie wyodrębnienie samego test, aby jego kod był w osobnej metodzie. Tej metodzie nadamy nazwę opisującą, co sprawdza ten test:

```
public static void Main(string[] args)
{
  // Oczekujemy iloczynu dwóch liczb przekazanych do aplikacji
  Multiplication_ShouldResultInAMultiplicationOfTwoPassedNumbers();
}

public void
Multiplication_ShouldResultInAMultiplicationOfTwoPassedNumbers()
{
  //Zakładając, że...
  var multiplication = new Multiplication(3,7);

  //Kiedy dzieje się coś takiego:
  var result = multiplication.Perform();

  //Wtedy wynik powinien być taki...
  AssertTwoIntegersAreEqual(expected: 21, actual: result);
}

public static void AssertTwoIntegersAreEqual(
  int expected, int actual)
{
  // Sprawdzamy, że podane liczby całkowite (w tym przypadku oczekiwana i zwrócona) \
są sobie równe.
  if(actual != expected)
  {
    throw new Exception(
    "Failed! Expected: " + expected + " but was: " + actual);
  }
}
```

I to wszystko. Teraz, jeśli potrzebujemy kolejnego testu, np. dla dzielenia, możemy po prostu dodać kolejne wywołanie, innej metody testujacej do `Main()` a następnie zaimplementować tę metodę. Wewnątrz nowego testu możemy ponownie użyć metody `AssertTwoIntegersAreEqual()`, ponieważ sprawdzenie wyników dzielenia będzie również opierało się na porównania dwóch wartości całkowitych - oczekiwanej i tej faktycznie zwróconej.

Jak widzisz, możemy łatwo napisać zautomatyzowane testy, używając naszych prymitywnych metod. Takie podejście ma jednak pewne wady:

1. Za każdym razem, gdy dodajemy nowy test, musimy zaktualizować metodę `Main()` o wywołanie nowego testu. Jeśli zapomnimy tego, test nigdy nie zostanie uruchomiony. Na początku nie jest to wielka sprawa, ale gdy już będziemy mieć dziesiątki testów, trudno będzie zauważyć te niedodane.

2. Wyobraź sobie, że Twój system składa się z więcej niż jednej aplikacji - miałbyś problemy ze zbieraniem wyników testów z wszystkich aplikacji, z których składa się twój system.
3. Wkrótce będziesz musiał napisać wiele innych metod podobnych do `AssertTwoIntegersAreEqual()` – ta tutaj porównuje dwie liczby całkowite, ale co jeśli chcemy sprawdzić inny warunek, np. czy jedna liczba całkowita jest większa od innej? Co by było, gdybyśmy chcieli sprawdzić równość nie dla liczb całkowitych, ale dla znaków, ciągów znaków, itp.? Co by było, gdybyśmy chcieli sprawdzić pewne właściwości kolekcji, np. czy kolekcja jest posortowana, lub czy wszystkie elementy w kolekcji są unikatowe?
4. Jeśli test się nie powiedzie, trudno będzie przenieść się od komunikatu na konsoli do odpowiedniego wiersza w kodzie źródłowym w twoim IDE. Czy nie byłoby łatwiej - kliknąć komunikat o błędzie i zostać przeniesionym do miejsca w kodzie, dzie wystąpił błąd?

Z tego względu i kilku innych, stworzono zaawansowane, zautomatyzowane narzędzia do testowania aplikacji - frameworki testujące, takie jak CppUnit (dla C++), JUnit (dla Javy) lub NUnit (C#). Frameworki testujące są w zasadzie oparte na tej samej idei, którą opisałem powyżej, ale jednocześnie nadrabiają wady naszego wcześniejszego, prymitywnego podejścia. Struktura i funkcjonalność tych framework'ów wywodzą się ze Smalltalk's SUnit, są określane jako rodzina testów **xUnit**.

Szczerze mówiąc, nie mogę się doczekać, by pokazać Ci jak będzie wyglądać nasz wcześniejszy test, napisany przy użyciu frameworka testujacego. Jednakże najpierw podsumujmy to, co się nam udało osiągnąć do tej pory. Wprowadźmy też pewną terminologię, która pomoże nam zrozumieć, w jaki sposób zautomatyzowane frameworki testujące rozwiązują nasze problemy:

1. Metoda `Main()` posłużyła nam jako lista testów (**Test List**) - miejsce, w którym decyduje się, które testy należy uruchomić.
2. Metoda `Multiplication_ShouldResultInAMultiplicationOfTwoPassedNumbers()` była naszą metodą testową (**Test Method**).
3. Metoda `AssertTwoIntegersAreEqual ()` jest asercją (**Assertion**) - warunkiem, który - gdy nie zostanie spełniony - kończy test niepowodzeniem.

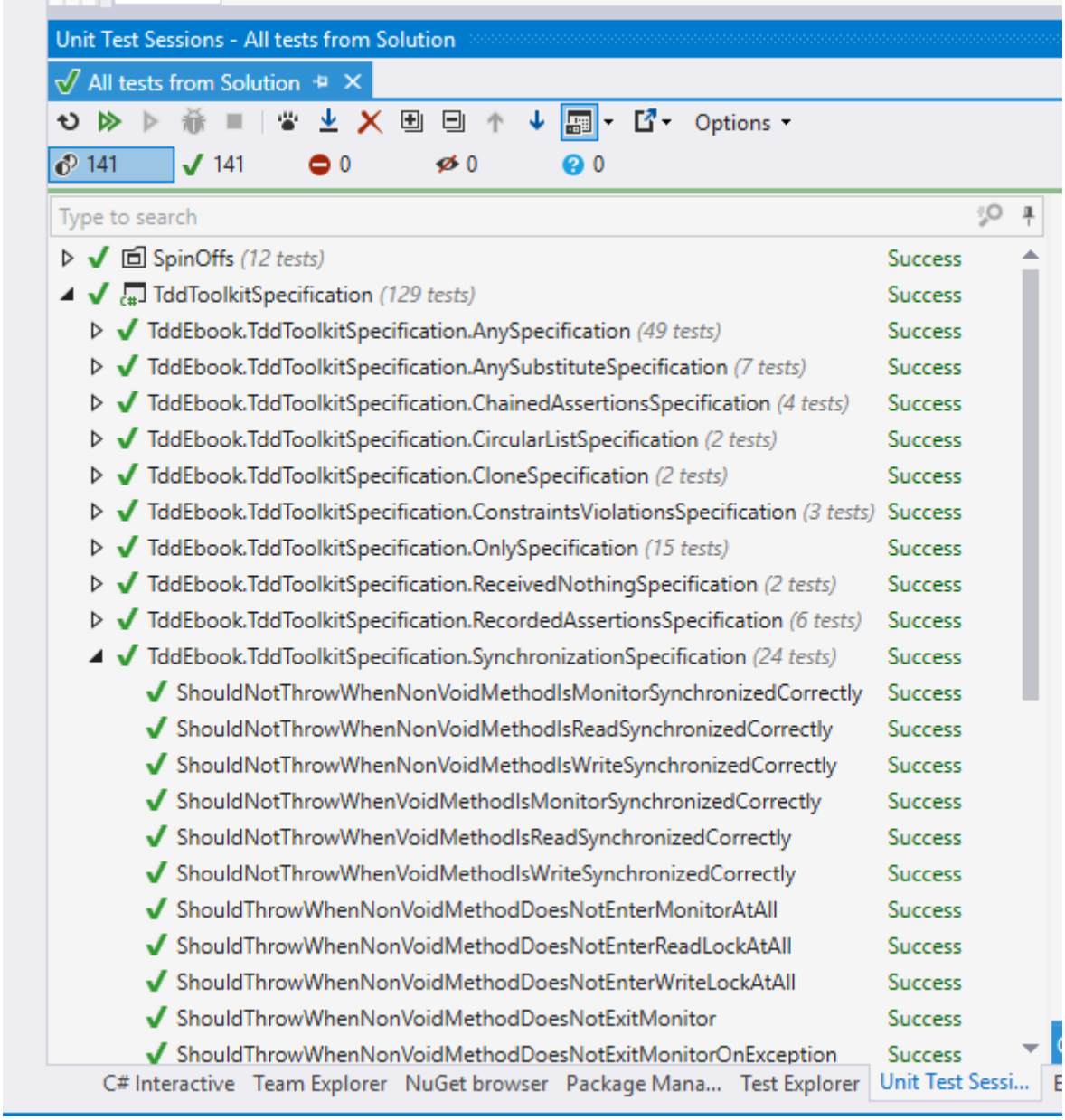
Ku naszej radości, te trzy chwalebne elementy są również obecne, gdy używamy frameworka testującego. Ponadto są znacznie bardziej zaawansowane. Aby to zilustrować, oto (nareszcie!) ten sam test, który napisaliśmy powyżej, teraz używający frameworka testowego [xUnit.Net] (http://xunit.github.io/):

```
[Fact] public void
Multiplication_ShouldResultInAMultiplicationOfTwoPassedNumbers()
{
  //Zakładając, że...
  var multiplication = new Multiplication(3,7);

  //Kiedy dzieje się to:
  var result = multiplication.Perform();

  //Wtedy wyniki powinny być takie...
  Assert.Equal(21, result);
}
```

Patrząc na przykład widzimy, że metoda testu jest jedyną rzeczą, która pozostała - lista testów i asercja, które poprzednio mieliśmy, zniknęły. Cóż, prawdę mówiąc, one nie do końca znikają - po prostu framework testujący oferuje zastępstwa, które są o wiele lepsze - więc ich użyliśmy. Odświeżmy sobie trzy elementy z poprzedniej wersji testu, o których mówiłem, że teraz również będą obecne:

1. Lista testów (**Test List**) jest teraz tworzona automatycznie przez framework na podstawie wszystkich metod oznaczonych atrybutem `[Fact]`. Nie ma potrzeby zarządzać z poziomu kodu już żadnymi listami, dlatego znika metoda `Main()`.
2. Metoda testująca (**Test Method**) wciąż jest obecna i wygląda niemalże tak jak wcześniej.
3. Asercja (**Assertion**) przyjęła kształt statycznej metody `Assert.Equal()` – xUnit.NET framework posiada szeroki zakres takich asercji, a więc użyłem jednej z nich. Oczywiście, nie ma przeszkód byś napisał swoją własną asercję, jeśli framework nie oferuje Ci tego, czego szukasz.

Uff, mam nadzieję, że to przejście do frameworka testującego okazało się w miarę bezbolesne dla Ciebie. Teraz ostatnia rzecz - skoro nie ma już metody `Main()`, to pewnie się zastanawiasz jakim cudem uruchamiamy te testy, prawda? Dobrze, wyjawię Ci ostatni sekret – używamy zewnętrzenej aplikacji do tego celu (*po polsku można ją nazwać odpalaczem testów, po angielsku to* **Test Runner**) – określamy które zestawy z testami (assemblies) chcemy załadować, rest runner uruchamia testy, tworzy raporty na podstawie wyników etc. Nasz odpalacz może przyjać wiele form, to może być aplikacja konsolowa, aplikacja z GUI albo plugin do IDE. Oto przykład test runner'a dostarczanego jako plugin do Visual Studio IDE, nazywającego się Resharper:



Resharper test runner docked as a window in Visual Studio 2015 IDE

Framework do mockowania

To wprowadzenie jest przeznaczone dla tych, którzy nie są biegli w używaniu mocków (*czytaj: "moków", mock to kolejny angielskojęzyczny termin w informatyce, który nie ma swojego odpowiednika w języku polskim - dosłownie tłumacząc z angielskiego, mock to imitacja*). Mocki mogą nie być najłatwiejsze do zrozumienia i dlatego jestem w stanie zaakceptować, jeśli na razie będziesz miał problemy z uchwyceniem tej koncepcji. Jeśli, podczas czytania tego wprowadzenia, zgubisz się - nie zwracaj na to w ogóle uwagi i idź dalej. Będziemy zajmować się mocno mockami w drugiej części książki, gdzie zagwarantuję Ci bogatszy i dokładniejszy opis.

Kiedy chcemy przetestować klasę, która zależy od innej klasy, możnaby sądzić, że dobrym pomysłem jest umieszczenie w teście również tej drugiej klasy. To, jednakże, nie pozwoli nam testować wyłącznie jednego obiektu, czy też małej grupy obiektów tak, byśmy mogli sprawdzić prawidłowe działanie najmniejszego fragmentu aplikacji. *Testujemy najmniejszy fragment programu, bo gdy test nie będzie przechodził, łatwiej będzie znaleźć w małym fragmencie miejsce i przyczynę wystąpienia błędu.* Jeśli sprawimy, iż nasza klasa nie zależy od innych klas, ale zależy raczej od abstrakcji w postaci interfejsów - możemy z łatwością implementować te interfejsy za pomocą specjalnych, "fałszywych" klas, stworzonych w taki sposób, by ułatwić nam testowanie. Na przykład obiekty takich klas mogą zawierać wstępnie zaprogramowane wartości zwracane dla metody zadeklarowanej w interfejsie. Mogą także zapamiętywać, które metody były wywoływane i zezwalać testowi na sprawdzenie, czy komunikacja między obiektem poddawanym testowi a jego zależnościami jest poprawna.

To może nie mieć znaczenia w Twoim przypadku, ale preferowanym podejściem jest stworzenie imitacji obiektu na podstawie interfejsu, a nie klasy, ponieważ normalnie, jeśli podążasz za TDD (Test Driven Development), możesz napisać testy jednostkowe jeszcze przed napisaniem implementacji zależnych klas. Dlatego, nawet jeśli nie masz konkretnej klasy DataAccessImpl, nadal możesz używać interfejsu DataAccess.

Niekorzystanie z interfejsów skutecznie utrudnia TDD, bo zmusza nas do tworzenia zależnych klas wraz z ciałami metod, nawet wtedy, kiedy ich jeszcze nie potrzebujemy. Żeby skomplikować sprawę, dodam - że w Javie można, bez przeszkód, stworzyć imitacje na podstawie definicji klasy - nie da się tego samego równie bezproblemowo zrobić w C#. Warto zaznaczyć, że interfejs w C# nigdy nie będzie zawierał metod.

Co więcej, frameworki do mockowania (imitowania) mają ograniczenia w imitowaniu klas, a niektóre frameworki pozwalają tylko na imitowanie interfejsów.

W dzisiejszych czasach możemy zdać się na narzędzia do generowania takiej "fałszywej" implementacji danego interfejsu, co pozwoli nam wykorzystać tę wygenerowaną implementację zamiast prawdziwego obiektu w testach. Dzieje się to na różny sposób, w zależności od języka. Czasami implementacje interfejsu mogą być generowane w czasie wykonywania (tak jak w Javie lub C#), czasami musimy polegać bardziej na generowaniu w czasie kompilacji (np. w C++).

Zawężając sprawę do samego C# - framework mockujący jest mechanizmem, który pozwala nam tworzyć obiekty-imitacje (zwane "mockami"), które kojarzone są z interfejsem, w czasie wykonywania. Działa to tak: typ interfejsu, który chcemy zaimplementować, jest zwykle przekazywany do specjalnej metody, która zwraca obiekt `mock` oparty na tym interfejsie (zobaczymy przykład w kilka sekund). Oprócz tworzenia imitacji obiektów, taki framework zapewnia interfejs API do określania, jak mocki powinny sue zachowywać podczas wywyływania określonych metod. To API pozwala nam również sprawdzić, które metody zostały wywołane. Jest to bardzo ważna funkcja, ponieważ możemy zasymulować takie sytuację lub zweryfikować takie warunki początkowe, które byłyby trudne do osiągnięcia przy użyciu kodu produkcyjnego. Frameworki do mockowania nie są tak stare jak frameworki do testowania, więc nie były używane w TDD od samego początku.

Teraz pokażę Ci krótki przykład użycia frameworka mockującego, natomiast przełożę dalsze wyjaśnienie do późniejsztch rozdziałów, ponieważ pełny opis mocków i ich miejsca w TDD nie jest taki łatwy do przekazania.

Załóżmy, że mamy klasę, która umożliwia składanie zamówień, a następnie umieszcza te zamówienia w bazie danych (za pomocą implementacji interfejsu o nazwie `OrderDatabase`). Dodatkowo, klasa obsługuje wszelkie wyjątki, które mogą wystąpić i zapisuje je do logu. Klasa sama w sobie nie robi żadnych ważnych rzeczy, ale spróbujmy wyobrazić sobie naprawdę mocno, że to ważna logika domenowa. Oto kod dla tej klasy:

```
public class OrderProcessing
{
  OrderDatabase _orderDatabase; // OrderDatabase to interfejs
  Log _log;

  // pobieramy obiekt bazodanowy spoza klasy:
  public OrderProcessing(
    OrderDatabase database,
    Log log)
  {
    _orderDatabase = database;
    _log = log;
  }

  public void Place(Order order)
  {
    try
    {
      _orderDatabase.Insert(order);
    }
    catch(Exception e)
    {
      _log.Write("Could not insert an order. Reason: " + e);
```

```
24      }
25    }
26
27     // reszta kodu...
28  }
```

Teraz wyobraź sobie, że musimy to przetestować – jak to robimy? Już widzę, jak potrząsasz głową i mówisz: "Stwórzmy połączenie z bazą danych, wywołajmy metodę `Place()` i sprawdźmy, czy rekord jest poprawnie dodany do bazy danych". Jeśli to zrobimy, pierwszy test będzie wyglądał następująco:

```
1   [Fact] public void ShouldInsertNewOrderToDatabaseWhenOrderIsPlaced()
2   {
3     //GIVEN
4     var orderDatabase = new MySqlOrderDatabase(); //użycie prawdziwej bazy danych
5     orderDatabase.Connect();
6     orderDatabase.Clean(); //posprzątaj po poprzednim teście
7     var orderProcessing = new OrderProcessing(orderDatabase, new FileLog());
8     var order = new Order(
9       name: "Grzesiek",
10      surname: "Galezowski",
11      product: "Agile Acceptance Testing",
12      date: DateTime.Now,
13      quantity: 1);
14
15    //WHEN
16    orderProcessing.Place(order);
17
18    //THEN
19    var allOrders = orderDatabase.SelectAllOrders();
20    Assert.Contains(order, allOrders);
21  }
```

Na początku testu otwieramy połączenie z bazą danych i czyścimy wszystkie istniejące w niej zamówienia (więcej o tym wkrótce), następnie tworzymy obiekt zamówienia, wstawiamy go do bazy danych, a potem pobieramy wszystkie zamówienia z bazy. Na koniec sprawdzamy, czy zamówienie, które próbowaliśmy wprowadzić, znajduje się wśród wszystkich zamówień.

Dlaczego czyścimy bazę danych na początku testu? Pamiętaj, że baza danych trwale przechowuje dane. Jeśli nie wyczyścimy ich przed wykonaniem logiki testu, baza danych może już zawierać element, który próbujemy dodać, np. z poprzedniego wykonania się testu. Co więcej, baza danych może nie pozwolić nam na ponowne dodanie tego samego produktu, a test zakończy się niepowodzeniem. Ałaaa! To tak bardzo boli - chcieliśmy, aby nasze testy udowodniły, że coś działa, ale wygląda na to, że mogą zawieść nawet wtedy, gdy logika jest poprawnie zakodowana. Jakie zastosowanie

miałby taki test, gdyby nie mógł nam odpowiedzieć na pytanie, czy zaimplementowana logika jest poprawna czy nie? Tak więc, aby upewnić się, że stan bazy danych jest taki sam za każdym razem, gdy uruchamiamy test, przed każdym uruchomieniem czyścimy bazę danych.

Czy teraz, kiedy test jest gotowy, dostaliśmy to, czego chcieliśmy? Wahałbym się, czy odpowiedzieć "tak". Jest kilka powodów:

1. Test będzie najprawdopodobniej wykonywał się wolno, ponieważ dostęp do bazy danych jest stosunkowo wolny. Nierzadko zdarza się, że w projekcie do wykonania jest ponad tysiąc testów. Nie chcę za każdym razem czekać pół godziny na wyniki, gdy uruchamiam testy jednostkowe. A ty?
2. Każdy, kto chce uruchomić ten test, musi skonfigurować specjalne środowisko, np. lokalną baza danych na swoim komputerze. Co, jeśli czyjaś konfiguracja różni się nieco od mojej? Co się stanie, jeśli schemat produkcyjnej bazy danych stanie się nieaktualny - czy każdy zdąży to zauważyć i zaktualizować schemat swoich lokalnych baz danych? Czy powinniśmy ponownie uruchomić nasz skrypt do tworzenia bazy danych tylko po to, aby upewnić się, że dysponujemy aktualnym schematem, dla którego można przeprowadzić testy?
3. Możemy nie być w stanie uruchomić systemu bazodanowego na naszym komputerze, jeśli docelowo ma on działać na jakiejś egzotycznej platformie, albo na urządzeniu mobilnym.
4. Zauważ, że test, który napisaliśmy, jest tylko jednym z dwóch nam potrzebnych. Nadal musimy napisać kolejny test dla scenariusza, gdzie wstawienie zamówienia kończy się rzuceniem wyjątku. Jak skonfigurować bazę danych, by rzuciła wyjątek? Jest to możliwe, ale wymaga znacznego wysiłku (np. usunięcie tabeli i odtworzeniu jej po teście - bo inne testy mogą wymagać tej tabeli do prawidłowego działania). To może doprowadzić nas do wniosku, że nie warto pisać testów w ogóle.

Teraz spróbujmy podejść do tego problemu w inny sposób. Załóżmy, że klasa `MySqlOrderDatabase` wysyłająca zapytania do bazy danych, jest już przetestowana (nie chcę jeszcze wdawać się w dyskusję na temat testowania zapytań do bazy danych - dojdziemy do tego w późniejszych rozdziałach) i załóżmy, że jedyną rzeczą, którą musimy przetestować, jest klasa `OrderProcessing` (pamiętajcie, staramy się naprawdę mocno wyobrazić, że jest tu zakodowana pewna poważna logika domenowa). W tej sytuacji możemy usunąć `MySqlOrderDatabase` z testu i zamiast tego stworzyć fałszywą implementację `OrderDatabase`. Będzie ona działała tak, jakby wykonywała prawdziwe połączenie z bazą danych, ale nie będzie w ogóle zapisywała tam informacji (zapisze wstawione rekordy na liście, w pamięci RAM komputera). Kod takiego udawanego połączenia może wyglądać tak:

```
public class FakeOrderDatabase : OrderDatabase
{
  public Order _receivedArgument;

  public void Insert(Order order)
  {
    _receivedArgument = order;
  }

  public List<Order> SelectAllOrders()
  {
    return new List<Order>() { _receivedOrder };
  }
}
```

Zauważ, że klasa imitująca połączenie z bazą danych jest instancją klasy implementującą ten sam interfejs co `MySqlOrderDatabase`. Tak więc, możemy sprawić, że testowany kod użyje fałszywej bazy danych nawet o tym nie widząc.

Zastąpmy, w naszym teście, prawdziwe połączenia z bazą danych fałszywą implementacją:

```
[Fact] public void
ShouldInsertNewOrderToDatabaseWhenOrderIsPlaced()
{
  //GIVEN
  var orderDatabase = new FakeOrderDatabase();
  var orderProcessing = new OrderProcessing(orderDatabase, new FileLog());
  var order = new Order(
    name: "Grzesiek",
    surname: "Galezowski",
    product: "Agile Acceptance Testing",
    date: DateTime.Now,
    quantity: 1);

  //WHEN
  orderProcessing.Place(order);

  //THEN
  var allOrders = orderDatabase.SelectAllOrders();
  Assert.Contains(order, allOrders);
}
```

Zauważ, że nie czyścimy obiektu fałszywej bazy danych, tak jak robiliśmy to z prawdziwą bazą danych, ponieważ tworzymy nowy obiekt za każdym razem, gdy test jest uruchamiany, a wyniki są

przechowywane w innym miejscu pamięci dla każdej instancji. Test będzie teraz znacznie szybszy, ponieważ nie mamy już dostępu do prawdziwej bazy danych. Co więcej, możemy teraz łatwo napisać test na wypadek błędu przy dodawaniu nowego zamówienia. W jaki sposób? Po prostu zaimplementujemy kolejne połączenie z nieprawdziwą bazą danych w ten sposób:

```
public class ExplodingOrderDatabase : OrderDatabase
{
  public void Insert(Order order)
  {
    throw new Exception();
  }

  public List<Order> SelectAllOrders()
  {
  }
}
```

Ok, na razie dobrze, ale teraz mamy dwie klasy połączeń z fałszywą bazą danych do utrzymania (i są szanse, że będziemy potrzebować ich jeszcze więcej). Każda metoda dodana do interfejsu `OrderDatabase` musi również zostać dodana do każdej z tych fałszywych klas. Możemy zaoszczędzić trochę kodu, czyniąc nasze imitacje nieco bardziej generycznymi, byśmy ich zachowania mogli konfigurować za pomocą wyrażeń lambda:

```
public class ConfigurableOrderDatabase : OrderDatabase
{
  public Action<Order> doWhenInsertCalled;
  public Func<List<Order>> doWhenSelectAllOrdersCalled;

  public void Insert(Order order)
  {
    doWhenInsertCalled(order);
  }

  public List<Order> SelectAllOrders()
  {
    return doWhenSelectAllOrdersCalled();
  }
}
```

Teraz nie musimy tworzyć dodatkowych klas dla nowych scenariuszy, ale nasza składnia stała się bardziej uciążliwa. Oto jak konfigurujemy fałszywą bazę danych, by pamiętała i pozwalała odczytać wprowadzone zamówienie:

```
var db = new ConfigurableOrderDatabase();
Order gotOrder = null;
db.doWhenInsertCalled = o => {gotOrder = o;};
db.doWhenSelectAllOrdersCalled = () => new List<Order>() { gotOrder };
```

A jeśli chcemy rzucić wyjątek, gdy coś jest wstawiane:

```
var db = new ConfigurableOrderDatabase();
db.doWhenInsertCalled = o => {throw new Exception();};
```

Na szczęście niektórzy sprytni programiści stworzyli biblioteki, które zapewniają dalszą automatyzację w takich sytuacjach. Jedną z takich bibliotek jest [**NSubstitute**] (http://nsubstitute.github.io/). Zapewnia ona API w postaci metod rozszerzających (extension methods) C# - co może Ci się na początku wydawać się nieco magiczne, szczególnie jeśli nie znasz C#. Nie martw się, przyzwyczaisz się do tego.

Używając NSubstitute, nasz pierwszy test może zostać napisany w taki sposób:

```
[Fact] public void ShouldInsertNewOrderToDatabaseWhenOrderisPlaced()
{
  //GIVEN
  var orderDatabase = Substitute.For<OrderDatabase>();
  var orderProcessing = new OrderProcessing(orderDatabase, new FileLog());
  var order = new Order(
    name: "Grzesiek",
    surname: "Galezowski",
    product: "Agile Acceptance Testing",
    date: DateTime.Now,
    quantity: 1);

  //WHEN
  orderProcessing.Place(order);

  //THEN
  orderDatabase.Received(1).Insert(order);
}
```

Zauważ, że nie potrzebujemy już metody `SelectAllOrders()` w interfejsie do komunikacji z bazą danych. Istniała tylko po to, aby ułatwić pisanie testu - nie używał jej żaden kod produkcyjny. Możemy usunąć tę metodę i pozbyć się kolejnych problemów z utrzymaniem kodu. Zamiast wywoływania funkcji `SelectAllOrders()`, nasz mock utworzony przez NSubstitute zapisuje wywołania wszystkich swoich funkcji, pozwalając nam na skorzystanie ze specjalnej metody o nazwie

`Received()` (patrz ostatnia linia tego testu). W istocie, jest to zakamuflowana asercja sprawdzająca, czy metoda `Insert()` została wywołana z konkretnym obiektem zamówienia jako parametr.

To objaśnienie mocków jest bardzo płytkie, a jego celem jest tylko sprawienie, abyś zaczął działać. Wrócimy do mocków później, ponieważ zaledwie podrapaliśmy powierzchnię.

Generator wartości anonimizowanych

Patrząc na dane testowe z poprzedniej sekcji widzimy, że wiele wartości podano bardzo konkretnie, np. w następującym kodzie:

```
var order = new Order(
  name: "Grzesiek",
  surname: "Galezowski",
  product: "Agile Acceptance Testing",
  date: DateTime.Now,
  quantity: 1);
```

imię, nazwisko, produkt, data i ilość są bardzo specyficzne. Może to sugerować, że te konkretne wartości są ważne z punktu widzenia zachowania, które testujemy. Z drugiej strony, gdy ponownie spojrzymy na kod, który jest testowany:

```
public void Place(Order order)
{
  try
  {
    this.orderDatabase.Insert(order);
  }
  catch(Exception e)
  {
    this.log.Write("Could not insert an order. Reason: " + e);
  }
}
```

możemy zauważyć, że wartości te nie są nigdzie używane - testowana klasa nie wymaga ich, ani nie sprawdza w żaden sposób. Te wartości mogłyby być ważne z punktu widzenia bazy danych, ale już zdążyliśmy pozbyć się prawdziwej bazy danych z testów. Czy nie przeszkadza ci, że wypełniamy obiekt zamówienia tak wieloma wartościami, które nie mają związku z samą logiką testu i które zakłócają strukturę testu niepotrzebnymi szczegółami? Aby usunąć ten bałagan, wprowadźmy metodę o opisowej nazwie, tworzącą zamówienie ale ukrywającą szczegóły, których osoba czytająca test wcale nie potrzebuje:

```
[Fact] public void
ShouldInsertNewOrderToDatabase()
{
  //GIVEN
  var orderDatabase = Substitute.For<OrderDatabase>();
  var orderProcessing = new OrderProcessing(orderDatabase, new FileLog());
  var order = AnonymousOrder();

  //WHEN
  orderProcessing.Place(order);

  //THEN
  orderDatabase.Received(1).Insert(order);
}

public Order AnonymousOrder()
{
  return new Order(
    name: "Grzesiek",
    surname: "Galezowski",
    product: "Agile Acceptance Testing",
    date: DateTime.Now,
    quantity: 1);
}
```

Teraz jest znacznie lepiej. Nie tylko sprawiliśmy, że test był krótszy, ale również pokazaliśmy czytelnikowi, że wartości użyte do utworzenia zamówienia nie mają znaczenia z punktu widzenia przetestowanej logiki przetwarzania zamówień, są zanonimizowane. Dlatego też nazwa `AnonymousOrder()`.

Przy okazji, czy nie byłoby miło, gdybyśmy sami nie musieli anonimizować obiektów, ale mogli polegać na innej bibliotece, która by dla nas generowała obiekty już zanonimizowane? Niespodzianka, jest jedna! Nazywa się [** Autofixture **] (https://github.com/AutoFixture/AutoFixture). Jest to przykład tak zwanego generatora anonimizowanych wartości (choć jego twórca lubi mówić, że jest to również implementacja wzorca projektowego Konstruktor Danych Testowych (Test Data Builder), ale pomińmy tutaj tę dyskusję.

Po zmianie naszego testu tak, by używał biblioteki `AutoFixture` dochodzimy do tego:

```
private Fixture any = new Fixture();

[Fact] public void ShouldInsertNewOrderToDatabase()
{
  //GIVEN
  var orderDatabase = Substitute.For<OrderDatabase>();
  var orderProcessing = new OrderProcessing(orderDatabase, new FileLog());
  var order = any.Create<Order>();

  //WHEN
  orderProcessing.Place(order);

  //THEN
  orderDatabase.Received(1).Insert(order);
}
```

W tym teście używamy instancji klasy `Fixture` (która jest częścią `AutoFixture`) do tworzenia anonimizowanych wartości za pomocą metody o nazwie `Create()`. Tym samym, to pozwala nam usunąć metodę `AnonymousOrder()`, dzięki czemu konfiguracja testu jest krótsza.

Nieźle, co? AutoFixture ma wiele zaawansowanych funkcji, ale żeby wszystko było proste, chciałbym ukryć jego obecność za statyczną klasą o nazwie `Any`. Najprostsza implementacja takiej klasy wyglądałaby tak:

```
public static class Any
{
  private static any = new Fixture();

  public static T Instance<T>()
  {
    return any.Create<T>();
  }
}
```

W następnych rozdziałach zobaczymy wiele różnych metod klasy `Any`, a także pełne wyjaśnienie filozofii, która za tym stoi. Im dłużej używasz tej klasy, tym bardziej rozszerza się ona o nowe metody tworzenia niestandardowych obiektów.

Podsumowanie

W niniejszym rozdziale przedstawiono trzy narzędzia, których będziemy używać w tej książce, po opanowaniu których, Twoje wytwarzanie oprogramowania sterowane testami (test-driven

development) będzie szło Ci bardziej płynnie. Jeśli ten rozdział nie sprawił, żebyś uważał użycie tych trzech narzędzi za zasadne, nie martw się - zagłębimy się w filozofię stojącą za nimi w następnych rozdziałach. Na razie chcę tylko, żebyś zapoznał się z ich składnią. No dalej, pobierz te narzędzia z Internetu, uruchom je, spróbuj napisać coś prostego przy ich użyciu. Nie musisz jeszcze rozumieć ich pełnego celu, po prostu zacznij zabawę :-).

To nie (tylko) test

Czy rolą testu jest tylko "weryfikacja" lub "sprawdzenie", że oprogramowanie działa? Z pewnością jest to istotna część jego `runtime value`, tj. wartości, którą otrzymujemy kiedy uruchamiamy test. Jednakże, gdy ograniczymy naszą perspektywę jedynie do testów, możemy dojść do wniosku, że jedyną cenną rzeczą w przeprowadzaniu testu jest możliwość jego wykonania i wyświetlenia wyniku. Wartosć projektowanie lub pisanie testu sprowadza się do tego, że można go uruchomić, a czytelność testu ma zaś wartość tylko podczas debugowania. Czy tak jest w istocie?

W tym rozdziale będę przekonywał, że czynności polegające na projektowaniu, implementowaniu, kompilowaniu i czytaniu testu są bardzo ważne. Pozwalają traktować testy jako coś więcej niż tylko "automatyczną kontrolę".

Kiedy test staje się czymś więcej

Studiowałem w Łodzi, dużym mieście w centrum Polski. Jak (zapewne) wszyscy inni studenci we wszystkich krajach świata, mieliśmy wykłady, ćwiczenia i egzaminy. Egzaminy były dość trudne. Ponieważ moja grupa informatyczna była na Wydziale Elektroniki i Elektrotechniki, musieliśmy zrozumieć wiele przedmiotów, które nie miały nic wspólnego z programowaniem. Na przykład: elektrotechnikę, fizykę ciała stałego lub metrologię elektryczną i elektroniczną.

Wiedząc, że egzaminy były trudne i że trudno było nauczyć się wszystkiego w trakcie semestru, wykładowcy czasami dostarczali nam przykładowe egzaminy z poprzednich lat. Pytania różniły się od tych na naszych egzaminach, ale struktura i rodzaje zadawanych pytań (praktyka i teoria itp.) były podobne. Zazwyczaj otrzymywaliśmy przykładowe pytania, zanim nauka stawała się naprawdę ciężka (co zwykle miało miejsce pod koniec semestru). Zgadnij, co się wtedy działo? Jak możecie podejrzewać - nie korzystaliśmy z testów, które otrzymaliśmy, tylko po to, by "zweryfikować" lub "sprawdzić" naszą wiedzę po ukończeniu nauki. Wręcz przeciwnie - zbadanie tych testów było pierwszym krokiem naszego przygotowania. Dlaczego tak było? Jaki był cel patrzenia na testy, skoro wiedzieliśmy, że i tak nie znamy większości odpowiedzi?

Chociaż myślę, że moi wykładowcy nie zgodziliby się tutaj ze mną, to mam dość zabawne spostrzeżenie, że to co robiliśmy było podobne do "Lean Software Development". Lean to filozofia gdzie kładzie się nacisk na pozbywanie się tego, co niepotrzebne (unikanie marnotrawstwa). Każda funkcjonalność lub produkt, które nie są nikomu potrzebne, są uważane za stratę, marnotrawstwo. To dlatego, że jeśli coś nie jest potrzebne teraz to nie ma najmniejszego powodu by założyć, że kiedykolwiek będzie potrzebne. Cała taka funkcjonalność lub produkt nie dodaje żadnej wartości biznesowej. Nawet jeśli kiedykolwiek *będzie* potrzebne, to - bardzo prawdopodobne - że i tak będzie wymagać pracy, aby dopasować to do potrzeb klienta. W takim przypadku, praca - która została włożona w oryginalne rozwiązanie, a teraz wymagające adaptacji - jest marnotrawstwem.

To kosztowało, ale nie przyniosło korzyści (nie mówię o takich rzeczach jak demo dla klienta, ale gotowy, dopracowany produkt lub funkcjonalność).

Aby wyeliminować marnotrawstwo, zazwyczaj staramy się dodawać funkcjonalności których się od nas żąda, zamiast "wpychać" funkcjonalności do produktu w nadziei, że pewnego dnia staną się one przydatne. Innymi słowy, każda funkcja ma zaspokoić konkretną potrzebę. Jeśli nie, pracę uważa się za zmarnowaną, a pieniądze poszły w błoto.

Wracając do egzaminów - dlaczego podejście polegające na przejrzeniu przykładowych testów można uznać za "lean"? Załóżmy, że naszym celem jest zaliczenie egzaminu. Dlatego - wszystko co nie przybliża nas do tego celu, jest uważane za marnotrawstwo. Jeśli egzamin z przedmiotu dotyczył tylko teorii - po co było przed egzaminem zajmować się ćwiczeniami? To, jaki jest egzamin, można było uzyskać na podstawie przykładowych testów. Testy były więc swoistą specyfikacją tego, co było potrzebne do zdania egzaminu. Pozwoliły nam uzyskać wartościowe informacje (np. wiedzę o kształcie egzaminu) na podstawie wymagań (informacji uzyskanych z prawdziwych testów), zamiast zmuszać nas do przeszukiwania wszystkiego, co już istniało (tj. próbować przeczytać wszystko, co się da pod kątem i ćwiczeń, i teorii).

Dlatego też testy okazały się czymś więcej, niźli tylko testami. Okazały się bardzo cenne jeszcze przed "implementacją" (tj. uczeniem się do egzaminu). Stało się tak, bo:

1. pomogły nam skupić się na tym, co było potrzebne, aby osiągnąć nasz cel
2. odciągnęły naszą uwagę od tego, co **nie** było konieczne, aby osiągnąć nasz cel

Swoją drogą, nie muszę chyba mówić, że celem nauki nie powinno być wyłącznie zdanie egzaminu?

Taka właśnie była wartość testu przed nauką. Zwróć uwagę, że testy, które otrzymywaliśmy, nie były dokładnie takie same, jak te w czasie egzaminu, więc nadal musieliśmy zgadywać. Jednak rola **testu jako wyszczególnienia potrzeby**, **wymagania** była już widoczna. *Można powiedzieć, rola testu jako specyfikacji wymagań.*

Testy w świecie programistów

Wybrałem tę długą metaforę, aby pokazać, że napisanie "testu" jest innym sposobem określenia wymagań, potrzeb - i że myślenie w ten sposób o testach nie jest sprzeczne z intuicją. Przykład z testami z poprzednich lat, przed egzaminem, to coś wziętego z codziennego życia. Ta sama sytuacja ma miejsce w przypadku rozwoju oprogramowania. Weźmy następujący "test" i zobaczmy, jakie potrzeby on określa:

```
1 var reporting = new ReportingFeature();
2 var anyPowerUser = Any.Of(Users.Admin, Users.Auditor);
3 Assert.True(reporting.CanBePerformedBy(anyPowerUser));
```

(W tym przykładzie użyliśmy metody `Any.Of()`, która zwraca jakąś wartość z podanej listy. Chcemy powiedzieć: "podaj mi wartość, która jest albo `Users.Admin` albo `Users.Auditor`".)

Spójrzmy na te trzy (tylko!) linie kodu i wyobraźmy sobie, że kod produkcyjny, który sprawi, że test zakończy się sukcesem, jeszcze w ogóle nie istnieje. Czego możemy się nauczyć na podstawie tych trzech linii o tym, co kod produkcyjny musi zrobić? Wyliczaj ze mną:

1. Potrzebujemy czegoś, co cechuje możliwość raportowania.
2. Musimy używać pojęcia użytkowników i przywilejów.
3. Musimy używać koncepcji użytkownika zaawansowanego, który jest administratorem lub przeprowadza audyt.
4. Tak zwani `Power users`, czyli "użytkownicy mający moc", muszą mieć możliwość raportowania (pamiętaj, że nie określiliśmy, czy jacyś inni użytkownicy powinni lub nie powinni mieć możliwości korzystania z funkcji raportowania - potrzebowalibyśmy do tego osobnego "testu").

Ponadto, już jesteśmy po fazie projektowania interfejsu API (ponieważ test już używa jakiegoś API), który pasowałby do powyższych wymagań. Czy nie sądzisz, że to całkiem dużo informacji o funkcjach aplikacji - wnioskując po zaledwie trzech liniach kodu?

Raczej specyfikacja niż zbiór testów

Mam nadzieję, że teraz widzicie, że to, co nazywaliśmy "testem", może być również postrzegane jako rodzaj specyfikacji. Jest to również odpowiedź na pytanie, które zadałem na początku tego rozdziału (o to, co jest rolą testu)

W rzeczywistości test, napisany przed kodem produkcyjnym, ma następujące funkcje :

- projektowanie scenariusza - kiedy określamy nasze wymagania, podając konkretne przykłady zachowań, których się spodziewamy
- pisanie kodu testowego - kiedy określamy interfejs API, za pomocą którego chcemy wywoływać testowany kod
- kompilowanie - gdy otrzymujemy informację od kompilatora, że kod produkcyjny ma klasy i metody wymagane przez specyfikację, którą napisaliśmy. Jeśli nie, kompilacja zakończy się niepowodzeniem.
- wykonanie - kiedy otrzymujemy informację zwrotną od testu, czy kod produkcyjny zachowuje się tak jak opisano w specyfikacji. Jeśli nie, test powinien zakończyć się niepowodzeniem, fiaskiem (failed).
- czytanie - to miejsce, w którym wykorzystujemy już napisaną specyfikację, aby uzyskać wiedzę na temat tego, jak korzystać z kodu produkcyjnego.

Dlatego nazwa "test" to trochę za mało, by oddać co w TDD robimy. Moje odczucia są takie, że inna nazwa mogłaby być lepsza - stąd określenie *specyfikacja.*

Odkrycie, że testy pełnią rolę specyfikacji jest dość niedawne i nie ma jeszcze jednolitej terminologii. Niektórzy lubią nazywać proces używania testów jako specyfikacje. *Specyfikacja przez przykłady* (specification by example) żeby przekazać, że testy są przykładami, które pomagają określić i wyjaśnić rozwijaną funkcjonalność. Niektórzy używają terminu BDD (*Behavior-Driven Development*), aby podkreślić, że pisanie testów polega na analizowaniu i opisywaniu zachowań. Ponadto możesz napotkać różne nazwy dla niektórych elementów takiego podejścia, na przykład "test" może być określany jako "specyfikacja", "przykład", "opis zachowania", lub "opis zachowania", albo "fakt o systemie". Zresztą, widzieliśmy w rozdziale o narzędziach, że xUnit.NET Framework oznacza każdy test atrybutem `[Fact]`, sugerując, że stwierdzamy pojedynczy fakt o tworzonym kodzie. Przy okazji, xUnit.NET pozwala nam również na określenie teorii na temat naszego kodu, ale zostawmy ten temat na inny czas.

Wziąwszy pod uwagę różnorodność w terminologii, umówmy się tak: żeby być spójnym przez całą książkę, ustanowię konwencje nazewniczą, ale ostatecznie Tobie zostawiam prawo wyboru tego, jaka nazwa jest dla Ciebie najodpowiedniejsza. Powodem takiego podejścia do nazewnictwa jest pedagogika - nie próbuję stworzyć ruchu na rzecz lansowania określonych pojęć, nie chcę wynaleźć nowej metodyki, ani niczego podobnego - mam nadzieję, że konsekwnetne używanie w książce poniższej terminologii, pozwoli Ci spojrzeć na niektóre rzeczy inaczej. Zgadzamy się więc, że ze względu na tę książkę:

Specyfikacja Wymagania (Specification Statement), lub po prostu Wymaganie (Statement), wielką literą 'W'
: będzie używane zamiast słowa "test" i "metoda testowa" ("test method")

Specyfikacja (Specification) również wielką literą 'S'
: będzie używana zamiast słów "zestaw testów" ("test suite") i lista testów ("test list")

Wymaganie niespełnione (False Statement)
: będzie używane zamiast "niepowodzenie testu" ("failing test")

Wymaganie spełnione (True Statement)
: będzie używane zamiast "test, który przeszedł" ("passing test")

Od czasu do czasu będę powracał do "tradycyjnej" terminologii, ponieważ jest lepiej utrwalona w środowisku IT i ponieważ słyszeliście już jakieś inne terminy, które zdążyły się zadomowić w świadomości programistów. Z pewnością zastanawiacie się, jak należy je rozumieć w kontekście myślenia o testach jako specyfikacji.

Różnice między "wykonywalnymi" specyfikacjami i tymi "tradycyjnymi"

Użytkownik może być zaznajomiony ze specyfikacjami wymagań lub specyfikacjami projektowymi napisanymi prostym językiem angielskim lub innym językiem mówionym. Jednakże nasze specyfikacje różnią się od nich w kilku kwestiach. W szczególności specyfikacja, którą tworzymy pisząc testy:

1. Nie jest *całkowicie* narzucona nam z góry, tak jak wiele "tradycyjnych" specyfikacji (co nie znaczy, że jest pisana po stworzeniu kodu - więcej na ten temat w następnych rozdziałach).
2. Jest "wykonywalna" - można ją uruchomić, aby sprawdzić, czy kod jest zgodny ze specyfikacją, czy też nie. Zmniejsza to ryzyko wystąpienia nieścisłości w Specyfikacji i nieprzystawalności Specyfikacji do kodu produkcyjnego.
3. Jest napisana za pomocą kodu źródłowego, a nie w języku mówionym - co jest dobre, ponieważ struktura kodu i sformalizowany styl pozostawiają mniej miejsca na nieporozumienia. Jednakże, jest to także wyzwanie, ponieważ należy zachować szczególną ostrożność, by utrzymać czytelność Specyfikacji.

Programowanie poprzedzone wymaganiem (Wymaganie-najpierw)

Po co pisać specyfikację po fakcie?

Jedną z najbardziej znanych rzeczy na temat TDD jest to, że piszemy nieprzechodzący test, zanim w ogóle zaimplementujemy w kodzie potrzebne zachowanie. Ta koncepcja jest często nazywana "test-first development" i dla wielu osób wydaje się być dość kontrowersyjna.

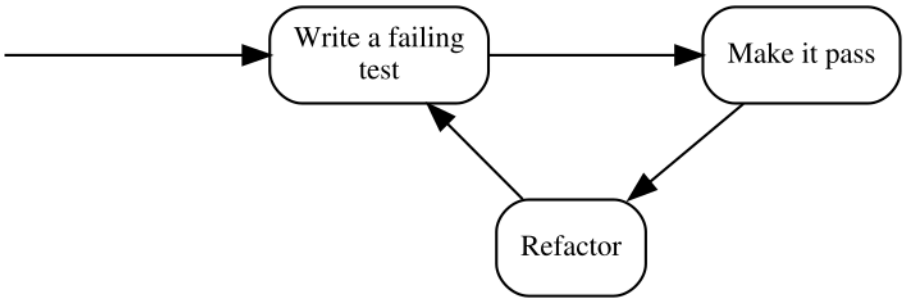
W poprzednim rozdziale powiedziałem, że w TDD "test" przyjmuje dodatkową rolę – wymagania, które jest częścią specyfikacji. Jeśli tak postawimy sprawę, cała kontrowersyjna koncepcja "pisania testu przed kodem" wcale nie stanowi problemu. Wręcz przeciwnie – wydaje się naturalne, aby sprecyzować, czego oczekujemy od kodu, zanim spróbujemy go napisać. Czy odwrotnie też to ma sens? Specyfikacja napisana po zakończeniu implementacji jest niczym więcej jak próbą udokumentowania istniejącego rozwiązania. Oczywiście, takie próby mogą stanowić jakąś wartość, gdy są wykonywane jako rodzaj inżynierii wstecznej (tj. gdy dokumentujemy istniejące rozwiązanie, albo zapisujemy specyfikację dla czegoś, co zostało zaimplementowane dawno temu i dla czego odkrywamy wcześniej niejawne reguły biznesowe). Pisanie specyfikacji ma w sobie coś z ekscytującego odkrywania, ale po tym, jak sami podjęliśmy wszystkie decyzje, nie wydaje mi się, aby był to produktywny sposób spędzania czasu. Nie wspominając o tym, że uważam to za nudne (możesz sprawdzić na własnej skórze, czy jesteś w stanie się ze mną zgodzić - spróbuj stworzyć prostą aplikację - kalkulator, a następnie napisz jej specyfikację zaraz po implementacji i ręcznym sprawdzeniu, czy działa). W każdym razie trudno mi doszukiwać się, jak coś powinno działać po tym jak to już zostanie stworzone. Może właśnie dlatego przez te lata zdążyłem zauważyć, że specyfikacje napisane "po" - są znacznie mniej kompletne niż te napisane przed wdrożeniem produktu.

Aha, i czy mówiłem wam, że bez jakiejkolwiek specyfikacji, nie wiemy, czy skończyliśmy wprowadzać zmiany do kodu, czy nie? Dzieje się tak, ponieważ aby ustalić, czy zmiana jest kompletna, musimy z "czymś" porównać zaimplementowaną funkcjonalność, nawet jeśli to "coś" znajduje się tylko w głowie klienta. w TDD "porównujemy" funkcjonalność z oczekiwaniami zaszytymi w zestawie automatycznych testów.

Inną rzeczą, o której wspomniałem w poprzednim rozdziale jest to, że zabieramy się za pisanie Specyfikacji za pomocą uruchamialnych Wymagań (Specification Statement) zupełnie inaczej niż gdy implementujemy, krok pro kroku, wygląd i działanie aplikacji korzystając z opisu, lub - chociażby - mamy z góry narzucone biznesowe wymagania (Requirements). W TDD, nawet jeśli zachowanie jest implementowane po tym, jak już istnieje koncepcja działania aplikacji, nie piszemy Specyfikacji tak, jakbyśmy mieli tekst przed oczyma i przekładali go na kod jota w jotę. Zazwyczaj postępujemy tak, że napiszemy trochę Specyfikacji, a potem trochę kodu aplikacji i tak w kółko. W

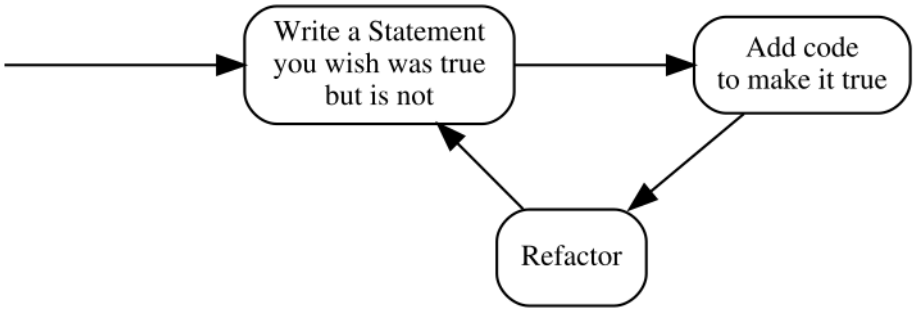
TDD przechodzimy wielokrotnie przez kilka faz, które składają się na cykl. Chcemy by te cykle były krótkie, abyśmy otrzymywali informacje zwrotne często i odpowiednio wcześnie. Informacje te są niezbędne, ponieważ pozwalają nam iść do przodu, dając pewność, że to co już mamy, działa zgodnie z naszymi zamierzeniami. Pozwalają nam również na usprawnienie następnego cyklu, dzięki wiedzy zdobytej w poprzednim cyklu (jeśli nie wierzysz, że liczy się szybka reakcja, zadaj sobie pytanie: "ile razy dziennie kompiluję kod, nad którym pracuję?").

Przeczytawszy tyle o cyklach, nie będzie dla Ciebie pewnie zaskoczeniem, że tradycyjna ilustracja procesu TDD jest wizualnie modelowana jako przepływ cykliczny:



Basic TDD cycle

Zwróć uwagę, że powyższy ilustarcja wykorzystuje tradycyjną terminologię TDD, więc zanim wyjaśnię kroki, oto podobna ilustracja, która korzysta z naszych pojęć Specyfikacji (Specifiction) i Wymagania (Statement):



Basic TDD cycle with changed terminology

1. Napisz Wymaganie, które chciałbyś, by było spełnione ale nie jest.
2. Dodaj kod, by Wymaganie zostało spełnione.
3. Zrefaktoruj kod.

Druga wersja wydaje się mieć bardziej sens niż pierwsza - określenie, jak coś powinno się zachowywać przed napisaniem kodu dla tego zachowania, jest bardziej intuicyjne niźli "testowaniu" czegoś, co jeszcze nie istnieje.

W każdym razie te trzy kroki zasługują na wyjaśnienie. W następnych rozdziałach podam kilka

przykładów na to, jak proces ten działa w praktyce i wprowadzę rozszerzoną wersję, ale w międzyczasie wystarczy wyjaśnić, że:

Napisz Wymaganie, które chciałbyś, by było spełnione ale nie jest.
: oznacza, że Wymaganie jest niespełnione i na liście testów pojawia się to jako błąd, fiasko (fail), który większość frameworków xUnit zaznacza kolorem czerwonym.

Dodaj kod, by Wymganie zostało spełnione.
: oznacza, że piszemy tylko tyle kodu, aby Wymaganie było spelnione, nie więcej. Na liście testów takie Wymaganie jawi się jako sukces (pass), które większość frameworków xUnit zaznacza kolorem zielonym. W dalszej części książki zobaczysz, jak mało może oznaczać "wystarczająco dużo".

Zrefaktoruj kod.
: jest krokiem, który do tej pory milcząco ignorowałem i zrobię to jeszcze przez kilka kolejnych rozdziałów. Nie martw się, w końcu wrócimy do tego. Na razie ważne jest, aby mieć świadomość, że Wykonywalna Specyfikacja może działać jak siatka zabezpieczając w cyrku. Podczas gdy my *poprawiamy jakość kodu bez zmiany jego zachowania zewnętrznego*, dzięki częstemu wykonywaniu kodu Specyfikacji, szybko odkrywamy każdy błąd, który popełniliśmy w procesie refaktorzacji.

Nawiasem mówiąc, proces ten jest czasami określany jako "Red-Green-Refactor", ze względu na kolory wyświetlane przez narzędzia xUnit w przypadku niepowodzenia i sukcesu testu. Tylko o tym tutaj wspominam – nie będę używał tego terminu w dalszej części książki.

"Najpierw-test" oznacza patrzenie na niepowodzenie

Wyjaśniając powyższy rysunek opisujący TDD, zwróciłem uwagę, że powinniśmy napisać Wymaganie, które chcielibyśmy, by było spełnione **ale nie jest**. Oznacza to, że nie tylko trzeba napisać Wymaganie przed implementacją, dzięki której wymaganie jest spełnione, ale musimy również to Wymaganie ewaluować (tj. uruchomić) i obserwować, że - w istocie - nie spełnia swoich założeń przed dostarczeniem implementacji.

Dlaczego to takie ważne? Czy nie wystarczy samo zapisanie Wymagania? Przecież wiadomo, że nie jest spełnione, gdy nie ma odpowiedniego kodu - po co je uruchamiać i oglądać jak świeci na czerwono na liście? Jest kilka powodów i postaram się pokrótce omówić kilka z nich.

Głównym powodem sprawdzenia, czy Wymaganie nie jest spełnione jest fakt, iż nie ma żadnego dowodu na to, że napisane Wymaganie skończy się kiedykolwiek spektakularnym fiaskiem po uruchomieniu.

Każde dobrze napisane Wymaganie nigdy nie przechodzi sprawdzenia, gdy nie jest spełnione, a zawsze przechodzi sprawdzenie, gdy jest spełnione. Jest to jeden z głównych powodów, dla których je piszemy - chcemy zobaczyć przejście od *Red* (czerwonego) do *Green* (zielonego), co oznacza, że to, co wcześniej nie zostało zaimplementowane nie działało (i mieliśmy na to dowód), a teraz działa (i również mamy na to dowód). Obserwacja przejścia Red-Green pokazuje, że zrobiliśmy postęp.

Inną rzeczą, na którą należy zwrócić uwagę, jest to, że napisanie kodu, który spełnia Wymaganie, sprawia iż staje się ono częścią uruchamialnej Specyfikacji. Gdy, choć przez chwilę, kod przestaje spełniać Wymaganie dowiadujemy się o tym (to może być na przykład wynik pomyłki podczas refaktoryzacji kodu).

Zauważenie, że Wymaganie nie jest spełnione dostarcza nam cennych informacji. Jeśli uruchomimy je wyłącznie *po* napisaniu kodu dla opisywanego zachowania, to skąd wiemy, że Wymaganie weryfikuje nasze prawdziwe oczekiwania i potrzeby? Nigdy nie widzieliśmy, by Wymaganie kiedykolwiek świeciło na czerwono, więc jaki mamy dowód, że w ogóle jest w stanie powiadomić nas o fiasku?

Pierwszy raz spotkałem się z tym argumentem, tuż przed tym, gdy zacząłem myśleć o testach jak o uruchamialnej specyfikacji. "Poważnie?" – pomyślałem – "Wiem doskonale, co koduje, jeśli zrobię wystarczająco małe testy, będzie oczywiste, że opisują prawidłowe zachowania. To jest zwykła paranoja". Jednak życie szybko zweryfikowało moje założenia i byłem zmuszony wycofać się ze swoich poglądów. Pozwolę sobie opisać trzy sposoby na to, jak napisać Wymaganie, które nigdy nie świeci na czerwono, niezależnie od tego, czy kod jest poprawny czy też nie. Tych sposobów jest więcej, ale myślę, że danie wam trzech powinno być wystarczającą ilustracją.

Oto sytuacje, kiedy odnosiłem wrażenie, że Wymaganie jest spełnione, nawet jeśli nie było:

1. Omyłkowe nieoznaczenie Wymagania w Specyfikacji

Zazwyczaj nie wystarczy napisanie kodu Wymagania - musimy także poinformować proces uruchamiający testy, że metoda, którą napisaliśmy, jest faktycznie Wymaganiem (a nie np. jakąś metodą pomocniczą) i musi zostać uruchomiona przez ten proces.

Większość frameworków typu xUnit ma jakiś mechanizm oznaczania metod jako Wymaganie, czy to przy użyciu atrybutów (C #, np. `[Fact]`) czy adnotacji (Java, np. `@ Test`), lub przy użyciu makr (C i C ++), lub przy użyciu konwencji nazewniczej. Musimy użyć takiego mechanizmu, aby proces uruchamiający testy wiedział, że powinien wykonywać takie metody.

Weźmy na przykład xUnit.Net. Aby przekształcić zwykłą metodę w Wymaganie, musimy oznaczyć ją za pomocą atrybutu `[Fact]`:

```
1  public class CalculatorSpecification
2  {
3    [Fact]
4    public void ShouldDisplayAdditionResultAsSumOfArguments()
5    {
6      //...
7    }
8  }
```

Jest szansa, że zapomnimy oznaczyć metodę atrybutem `[Fact]` - w takim przypadku ta metoda nigdy nie będzie wykonywana przez proces uruchamiający testy. Choć może to zabrzmieć zabawnie,

kilka razy mi się to przydarzyło. Wyobraźmy sobie, że powyższe Wymaganie piszemy post-factum jako test jednostkowy w środowisku, które ma, powiedzmy, ponad trzydzieści innych Wymagań już napisanych i spełnianych podczas uruchamiania. Napisaliśmy wcześniej kod dla naszych zachowań, a teraz dodajemy test po teście, aby upewnić się, że kod działa. Pierwszy test – sukces, drugi test – sukces, trzeci test – suckes... świetnie! Co ciekawe, kiedy wykonuję testy, prawie zawsze uruchamiam więcej niż jeden naraz (*służy do tego specjalny przycisk*), ponieważ jest to dla mnie łatwiejsze niż każdorazowe wybieranie z listy tego, co chcę aktualnie ewaluować. Poza tym, uruchamiając wszystkie testy, zyskuję więcej pewności, że nie popełniłem błędu i nie zepsułem czegoś, co było napisane wcześniej. Tu się pojawia problem, bo nawet jeśli każę uruchomić wszystkie testy i wszystkie przejdą, to te oznaczone nieprawidłwo nigdy nie zostaną uruchomione.

Z biegiem czasu nauczyłem się używać mechanizmu code snippets (*wstawianie szablonów kodu w edytorze*) w moim IDE do generowania szkieletu Wymagań. Wcześniej zdarzało mi się jednak pisać coś takiego:

```
1  public class CalculatorSpecification
2  {
3    //... jakieś inne Wymagania
4
5    //uups...zapomniałem wstawić atrybutu!
6    public void ShouldDisplayZeroWhenResetIsPerformed()
7    {
8      //...
9    }
10 }
```

Jak widzisz, brakowało atrybutu `[Fact]`, czyli Wymaganie nie było uruchamiane. Nawet nie dlatego, że nie umiałem korzystać z generatorów kodu - po prostu w celu utworzenia nowego Wymagania wygodniejsze było dla mnie skopiowanie i wklejenie innego Wymagania, zmiana jego nazwy i kilku linii jego kodu [^ copypaste]. Nie zawsze pamiętałem, aby dołączyć atrybut `[Fact]` w skopiowanym kodzie źródłowym. Kompilator też nie narzekał.

Powodem, dla którego nie potrafiłem dostrzec swojego błędu był fakt, że uruchamiałem cały czas wszystkie testy jednocześnie - i kiedy pojawił się zielony pasek (który oznacza, że wszystkie Wymagania są spełnione), założyłem, że nowonapisane Wymaganie również jest spełnione. Nie jest rzeczą fajną sprawdzanie, że każde nowe Wymaganie faktycznie pojawia się na liście Wymagań, więc tego nie robiłem. Co gorsza - brak atrybutu `[Fact]` nie zakłócał mojej pracy: pisałem test – wszystkie testy przeszły, pisałem kolejny test – wszystkie testy przeszły, kolejny test - wszystkie przeszły... Innymi słowy, mój sposób pracy nie dawał żadnej informacji zwrotnej, że popełniłem gdzieś, jakiś błąd. Tak więc, w tym przypadku, nie chodziło o to, że Wymaganie może nie być spełnione, bo jest źle napisane, ale o to, że **ono w ogóle nie zostało uruchomione i nie podlegało żadnej ewaluacji**.

W jaki sposób może pomóc postrzeganie testów jako Wymagań i uruchamianie ich przed napisaniem kodu dla konkretnego zachowania? W TDD normalny schemat wytawarzania programowania to:

test - fiasko - sukces, test - fiasko - sukces, test - fiasko - sukces... To podstawową różnica. Innymi słowy spodziewamy się, że podczas pracy w TDD, dowolne Wymaganie jest niespełnione conajmniej raz i sprawdzamy to. Tak więc za każdym razem, gdy nowy test nie kończy się "niepowodzeniem", otrzymujemy informację zwrotną że dzieje się coś mocno podejrzanego. To pozwala nam zacząć badać sprawę i rozwiązać problem.

2. Inicjalizacje umieszczone w złej kolejności

Dobrze, to może się wydawać jeszcze bardziej zabawne, ale zdarzyło mi się to kilka razy i zakładam, że pewnego dnia Tobie też może się przydarzyć, zwłaszcza jeśli się spieszysz.

Rozważmy następujący przykładzik: chcemy zweryfikować prostą strukturę danych, która odzwierciedla ramkę danych, które mogą dotrzeć przez sieć. Struktura wygląda następująco:

```
1  public class Frame /* ramka */
2  {
3    public int timeSlot; /* przedział czasu */
4  }
```

Musimy napisać specyfikację dla klasy `Validation` (*Walidacja*), która przyjmie obiekt `Frame` (*Ramkę*) jako argument, i sprawdzić, czy przedział czasowy (czymkolwiek on jest) jest poprawny. Poprawność określamy porównując przedział czasowy z maksymalną, dozwoloną wartością `TimeSlot.MaxAllowed` (to stała zdefiniowana w klasie `TimeSlot`). Jeśli przedział czasowy jest większy niż dopuszczalne maksimum, uważamy go za niepoprawny a walidacja powinna zwrócić `false` (*fałsz*). W innym przypadku powinno być zwrócone `true` (*prawda*).

Przyjrzyjmy się poniższemu Wymaganiu, które opisuje, że ustawienie wartości wyższej niż dozwolona dla pola `frame` powinno nie przejść walidacji:

```
1  [Fact]
2  public void ShouldRecognizeTimeSlotAboveMaximumAllowedAsInvalid()
3  {
4    var frame = new Frame();
5    var validation = new Validation();
6    var timeSlotAboveMaximumAllowed = TimeSlot.MaxAllowed + 1;
7    var result = validation.PerformForTimeSlotIn(frame);
8    frame.timeSlot = timeSlotAboveMaximumAllowed;
9    Assert.False(result);
10 }
```

Zwróć uwagę na metodę `PerformForTimeSlotIn()` wyzwalającą walidację, która omyłkowo została zawołana *przed* ustawieniem maksymalnej, dozwolonej wartości `timeSlotAboveMaximumAllowed` obiektu `frame`. Pożądana wartość nie jest w ogóle brana pod uwagę w momencie walidacji. Jeśli, na

przykład, popełnimy błąd podczas implementacji klasy `Validation` i, wbrew założeniom, walidacja będzie zwracała fałsz (`false`) dla wartości poniżej (a nie powyżej) wartości maksymalnej - błąd taki może pozostać niezauważony. Tak zapisane Wymaganie zawsze będzie spełnione.

To również jest trywialny przykład - użyłem go jako ilustrację czegoś, co da się przypadkowo popełnić, gdy ma się do czynienia z bardziej złożonymi przypadkami.

3. Używanie danych typu static wewnątrz kodu produkcyjnego

Od czasu do czasu musimy zajrzeć do Specyfikacji dodając kilka nowych Wymagań i trochę logiki do klas, które Specyfikacja opisuje. Załóżmy, że klasa i jej specyfikacja zostały napisane przez kogoś innego niż my, a wspomniany kod, jest klasą "opakowującą" dane z pliku konfiguracyjnego XML. Postanowiliśmy napisać nasze Wymaganie *po* wprowadzeniu nowych zmian (przecież możemy powiedzieć - "wszyscy jesteśmy chronieni przez Specyfikację, która już istnieje, więc można dokonać zmian bez ryzyka przypadkowego zniszczenia istniejącej funkcjonalności, a potem wystarczy przetestować zmiany i wszystko będzie w porządku...").

Zaczynamy sobie kodować... gotowe. Teraz piszemy nowe Wymaganie, które będzie opisywać właśnie dodaną funkcjonalność. Tymczasem, sprawdziwszy klasę Specyfikacji widzimy, że ma ona w sobie zaszyte takie oto pole:

```
1  public class XmlConfigurationSpecification
2  {
3    XmlConfiguration config = new XmlConfiguration(xmlFixtureString);
4
5    //...
```

Co się dzieje w tej linijce? Ustawiamy obiekt dostępny dla każdego Wymagania w Specyfikacji. Wkrótce okazuje się, że każde z Wymagań używa tego samego obiektu `config` zainicjalizowanego tą samą wartością ciągu `xmlConfiguration`. Kolejne szybkie sprawdzenie pozwala nam podejrzeć, co jest ukryte w `xmlFixtureString`:

```
1  <config>
2    <section name="General Settings">
3      <subsection name="Network Related">
4        <parameter name="IP">192.168.3.2</parameter>
5        <parameter name="Port">9000</parameter>
6        <parameter name="Protocol">AHJ-112</parameter>
7      </subsection>
8        <subsection name="User Related">
9        <parameter name="login">Johnny</parameter>
10       <parameter name="Role">Admin</parameter>
11       <parameter name="Password Expiry (days)">30</parameter>
```

```
    </subsection>
    <!-- i tak dalej, i tak dalej... -->
  </section>
</config>
```

Nasz łańcuch znaków jest już pokaźny i wypełniony różnymi, nie zawsze potrzebnymi rzeczami, ponieważ zawiera informacje wymagane przez wszystkie istniejące Wymagania. Załóżmy, że musimy napisać testy dla pewnych, skrajnych przypadków, które nie potrzebują tych wszystkich bzdurnych ustawień. Dlatego postanawiamy sobie stworzyć, od nowa, obiekt klasy `XmlConfiguration` zainicjalizowany naszym własnym ciągiem znaków o minimalnej długości. Początek Wymagania będziwe wyglądał tak:

```
string customFixture = CreateMyOwnFixtureForThisTestOnly();
var configuration = new XmlConfiguration(customFixture);
...
```

I uruchamiamy scenariusz testowy. Kiedy go wykonamy, zaświeci się na zielono, bo przeszedł - fajnie... nie, zaraz. Ok, co jest tu nie tak? Na pierwszy rzut oka wszystko jest w porządku, dopóki nie wczytamy się w kod źródłowy klasy XmlConfiguration. Wewnątrz widzimy, jak XML jest przechowywany:

```
private static string xmlText; //zwróć uwagę na słowo kluczowe static!
```

I tu jest pies pogrzebany. To pole statyczne, co oznacza, że jego wartość jest zachowywana między instancjami poszczególnych testów. Co takiego...? Już wyjaśniam, oto co się stało: autor tej klasy zastosował małą optymalizację. Pomyślał sobie tak: "W tej aplikacji konfiguracja jest modyfikowana tylko przez konsultantów-wdrożeniowców produktu, a żeby mogli ją trwale zmienić muszą i tak zamknąć system. Dlatego nie trzeba czytać pliku XML za każdym razem, gdy tworzony jest obiekt `XmlConfiguration`. Mogę zaoszczędzić kilka cykli procesora i kilka operacji wejścia/wyjścia, odczytując konfigurację tylko raz, gdy pierwsza instancja tej klasy zostanie utworzona. Przecież późniejsze obiekty będą używały tego samego pliku XML!". Fajnie dla niego, nie tak "spoko" dla nas. Czemu? Ponieważ, w zależności od kolejności, w jakiej ewaluowane są Wymagania, albo pole zostanie trwale zainicjowane tym przydługawym XML-em albo naszym-krótkim! W związku z tym Wymagania w tej Specyfikacji będą niedeterministycznie spełnione bądź nie z niewłaściwego powodu - gdy przypadkowo użyje się złego XML-a.

Rozpoczynanie pracy od napisania Wymagania, po którym oczekujemy, że nie będzie spełnione - pomoże nam w sytuacji, gdy Wymaganie niespodziewanie jest spełnione, chociaż nie zostało nawet zaimplementowane zachowanie przezeń opisywane.

"Test-Po" często kończy jako "Test-Nigdy"

Zastanów się ponownie nad pytaniem, które już zadałem w tym rozdziale: czy kiedykolwiek musiałeś napisać Wymagania lub dokument projektowy dla czegoś, co już zaimplementowałeś?

Fajnie było? Czy było to wartościowe? Czy to było twórcze? Jeśli chodzi o mnie, moją odpowiedzią na te pytania było - *nie*. Zauważyłem, że ta sama odpowiedź dotyczyła napisania przeze mnie wykonywalnej specyfikacji. Obserwując siebie i innych programistów, doszedłem do wniosku, że po napisaniu kodu mamy małą motywację do tworzenia specyfikacji dla tego, co napisaliśmy - widzimy, że "niektóre fragmenty kodu po prostu są poprawne", inne fragmenty "widzieliśmy, że działają" kiedy kompilowaliśmy kod i wdrażaliśmy nasze zmiany, przeprowadzając kilka ręcznych kontroli … Architektura aplikacji jest gotowa… Specyfikacja? Może następnym razem… W ten sposób, owa Specyfikacja nigdy nie zostaje napisana, a jeśli już - często widzę, że obejmuje tylko główne funkcje programu, ale brakuje w niej niektórych Wymagań mówiących o tym, co powinno się stać w przypadku błędów itp.

Kolejnym powodem, dla którego nie można napisać Specyfikacji, może być presja czasu, szczególnie w zespołach, które nie są jeszcze dojrzałe lub nie mają silnej etyki zawodowej. Wiele razy widziałem ludzi reagujących na naciski przez odrzucenie wszystkiego, co nie jest wyłącznie pisaniem kodu implementującego pożądane funkcję. Wśród rzeczy, które są odrzucane jest architektura kodu, wymagania i testy. I także nauka. Widziałem wiele razy zespoły, które pod presją przestały eksperymentować i uczyć się, a powróciły do starych "bezpiecznych" zachowań w sposobie myślenia "ratowanie tonącego statku" i "nadzieję na najlepsze". Jako, że w takich sytuacjach obserwowałem wzrost ciśnienia, gdy zbliżał się termin zakończenia lub osiągnięcie kamila milowego projektu, to wiem, że pozostawienie Specyfikacji na koniec oznacza, że prawdopodobnie całkowicie się z niej zrezygnuje, szczególnie w przypadku, gdy zmiany zostaną (do pewnego stopnia) przetestowane ręcznie później.

Z drugiej strony, kiedy robimy TDD (tak jak zobaczymy w kolejnych rozdziałach) nasza Specyfikacja rośnie wraz z kodem produkcyjnym, więc jest o wiele mniej pokusy, aby całkowicie z niej zrezygnować. Ponadto w TDD wyspecyfikowanie Wymagań nie jest dodatkiem do kodu, ale raczej *powodem* do napisania kodu. Tworzenie Wykonywalnej Specyfikacji staje się nieodzowną częścią dodawania nowych funkcji do programu.

"Test-Po" często prowadzi do ponownego projektowania

Lubię czytać i oglądać wuja Boba (Robert C. Martin). Pewnego dnia słuchałem jego przewodniego motywu na Ruby Midwest 2011, Architecture The Lost Years[3]. Na koniec Robert dokonał pewnych dygresji, z których jedna dotyczyła TDD. Powiedział, że pisanie testów po kodzie nie jest TDD i nazwał to "stratą czasu".

Moja pierwsza myśl była taka, że komentarz był chyba nieco zbyt przesadzony i dotyczył tylko braku korzyści wynikających z rozpoczęcia pracy nad kodem od niespełnionego Wymagania: kiedy widzimy niespełnione Wymaganie, można przeprowadzić niezakłóconą niczym analizę itp. Jednakże, teraz czuję, że chodzi o wiele więcej, a to za sprawą tego, czego nauczyłem się Amira Kolsky'ego i Scotta Baina - aby móc napisać przystepną w utrzymaniu Specyfikację dla jakiegoś

[3] http://www.confreaks.tv/videos/rubymidwest2011-keynote-architecture-the-lost-years

fragmentu kodu, kod musi mieć wysoki poziom **testowalności**. Porozmawiamy o jakości kodu w części drugiej tej książki, ale na razie przyjmijmy poniższą uproszczoną definicję: im większa testowalność kodu (np. klasy), tym łatwiej jest napisać Wymaganie określające jego zachowaniu.

Na czym polega strata czasu w pisaniu Specyfikacji po napisaniu kodu? Aby się tego dowiedzieć, porównajmy podejścia Najpierw-Wymaganie i Najpierw-Kod. W pierwszym przypadku dla nowopowstajacego (nie zastanego) mój przepływ pracy i podejście do testowalności zazwyczaj wyglądają tak:

1. Napisz Wymaganie, które jest niespełnione na początku (w tym kroku wykryj i popraw problemy z testowalnością jeszcze przed napisaniem kodu produkcyjnego).
2. Napisz kod, aby Wymaganie było prawdziwe.

A oto, co często robią programiści, gdy piszą najpierw kod (dodatkowe kroki oznaczone **pogrubionym tekstem**):

1. Napisz kod produkcyjny bez zastanawiania się, w jaki sposób zostanie on przetestowany (po tym etapie testowalność jest często nieoptymalna, ponieważ zwykle nie jest rozważana w tym momencie).
2. **Rozpocznij pisanie testu jednostkowego** (to może nie wydawać się dodatkowym krokiem, ponieważ jest również obecne w poprzednim podejściu, ale gdy dojdziesz do kroku 5 - będziesz wiedział co mam na myśli).
3. **Zauważ, że przy próbie napisania testu jednostkowego okazuje się, że kod sprawia trudność jeśli chodzi o testowanie, nie daje ku temu wielu możliwości, a pisane podeń testy zaczynają wyglądać strasznie nieprzejrzyście, ponieważ próbują obejść problemy z testowalnością.**
4. **Zdecyduj się poprawić testowalność poprzez restrukturyzację kodu, np. aby izolując obiekty i używając takich technik, jak mocki (imitacje)**
5. Napisz testy jednostkowe (tym razem powinno być łatwiej, ponieważ testowalność jest już lepsza).

Co jest odpowiednikiem pogrubonych wyżej kroków jeśli chodzi o pierwszy sposób? Nie ma żadnych odpowiedników! Robienie tych rzeczy to strata czasu! Niestety, jest to marnotrawstwo, z którym często się spotykam.

Podsumowanie

W tym rozdziale starałem się pokazać, że wybór *kiedy* piszemy naszą Specyfikację często robi ogromną różnicę i że istnieje wiele korzyści w zaczynaniu od Wymagania. Kiedy traktujemy Specyfikację jako zapis tego, co rzeczywiście dzieje się w programie - a nie tylko jako zestaw testów sprawdzających poprawność środowiska wykonawczego - wówczas podejście “Najpierw-Test” staje się mniej kłopotliwe i mniej sprzeczne z intuicją.

Poćwiczmy to, czego się właśnie nauczyliśmy

> And now, a taste of things to come! *A teraz przedsmak tego, co was czeka”*
>
> – Shang Tsung, Mortal Kombat The Movie

Cytowane słowa miały miejsce tuż przed sceną walki[4], w której bezimienny wojownik skoczył na Sub-Zero tylko po to, by zostać zamrożonym i rozbić się na kilka części po trafieniu w ścianę. Scena nie była spektakularna pod względem techniki walki ani długości. Ponadto, bezimienny gość nawet nie starał się walczyć - jedyną rzeczą, którą zrobił, był wyskok w powietrzę, by zostać uderzonym przez zmrożoną kulę, którą - nawiasem mówiąc - mógł dostrzec, gdy nadchodziła. Wyglądało to tak, jakby walka była ustawiona po to, by pokazać zdolność zamrażania Sub-Zero. I zgadnij co? W tym rozdziale zamierzamy zrobić z grubsza to samo - stworzyć fałszywy, łatwy scenariusz, aby pokazać niektóre z podstawowych elementów TDD!

Poprzedni rozdział był pełen teorii i filozofii, nie sądzisz? Naprawdę mam nadzieję, że nie zasnąłeś podczas czytania. Prawdę mówiąc, musimy przyswoić znacznie więcej tej teorii, dopóki nie będziemy w stanie pisać realnych aplikacji za pomocą TDD. Aby Ci to jakoś zrekompensować, proponuję dodatkową wycieczkę podczas naszej podróży, tylko po to, by wypróbować to, czego już się nauczyliśmy, na szybkim i łatwym przykładzie. Kiedy przezeń przejdziemy, możesz łamać sobie głowę, w jaki sposób moglibyśmy napisać prawdziwe aplikacje tak, jak się pisało nasz prosty program. Nie martw się, nie pokażę Ci jeszcze wszystkich trików, więc potraktuj to jako “przedsmak tego, co Cię czeka”. Innymi słowy, przykład będzie tak bliski problemom realnego świata, jak pojedynek między Sub-Zero i bezimiennym ninja był prawdziwą szkołą walki, jednakże pozwoli Ci to zobaczyć niektóre elementy procesu TDD.

Pozwól mi opowiedzieć sobie historię

Meet Johnny and Benjamin, two developers from Buthig Company. Johnny is quite fluent in programming and Test-Driven Development, while Benjamin is an intern under Johnny’s mentorship and is eager to learn TDD. They are on their way to their customer, Jane, who requested their presence as she wants them to write a small program for her. Along with them, we will see how they interact with the customer and how Benjamin tries to understand the basics of TDD. Like you, Benjamin is a novice so his questions may reflect yours. However, if you find anything explained in not enough details, do not worry – in the next chapters, we will be expanding on this material.

[4] https://www.youtube.com/watch?v=b0vhGEGJC8g

Akt 1: Samochód

Johnny: How do you feel about your first assignment?

Benjamin: I am pretty excited! I hope I can learn some of the TDD stuff you promised to teach me.

Johnny: Not only TDD, but we are also gonna use some of the practices associated with a process called Acceptance Test-Driven Development, albeit in a simplified form.

Benjamin: Acceptance Test-Driven Development? What is that?

Johnny: While TDD is usually referred to as a development technique, Acceptance Test-Driven Development (ATDD) is something more of a collaboration method. Both ATDD and TDD have a bit of analysis in them and work very well together as both use the same underlying principles, just on different levels. We will need only a small subset of what ATDD has to offer, so don't get over-excited.

Benjamin: Sure. Who's our customer?

Johnny: Her name's Jane. She runs a small shop nearby and wants us to write an application for her new mobile. You'll get the chance to meet her in a minute as we're almost there.

Akt 2: Tytuł dla Klienta

Johnny: Hi, Jane, how are you?

Jane: Thanks, I'm fine, how about you?

Johnny: Me too, thanks. Benjamin, this is Jane, our customer. Jane, this is Benjamin, we'll work together on the task you have for us.

Benjamin: Hi, nice to meet you.

Jane: Hello, nice to meet you too.

Johnny: So, can you tell us a bit about the software you need us to write?

Jane: Sure. Recently, I bought a new smartphone as a replacement for my old one. The thing is, I am really used to the calculator application that ran on my previous phone and I cannot find a counterpart for my current device.

Benjamin: Can't you just use another calculator app? There are probably plenty of them available to download from the web.

Jane: That's right. I checked them all and none has exactly the same behavior as the one I have used for my tax calculations. You see, this app was like a right hand to me and it had some really nice shortcuts that made my life easier.

Johnny: So you want us to reproduce the application to run on your new device?

Jane: Exactly.

Johnny: Are you aware that apart from the fancy features that you were using we will have to allocate some effort to implement the basics that all the calculators have?

Jane: Sure, I am OK with that. I got used to my calculator application so much that if I use something else for more than a few months, I will have to pay a psychotherapist instead of you guys. Apart from that, writing a calculator app seems like an easy task in my mind, so the cost isn't going to be overwhelming, right?

Johnny: I think I get it. Let's get it going then. We will be implementing the functionality incrementally, starting with the most essential features. Which feature of the calculator would you consider the most essential?

Jane: That would be addition of numbers, I guess.

Johnny: Ok, that will be our target for the first iteration. After the iteration, we will deliver this part of the functionality for you to try out and give us some feedback. However, before we can even deliver the addition feature, we will have to implement displaying digits on the screen as you enter them. Is that correct?

Jane: Yes, I need the display stuff to work as well – it's a prerequisite for other features, so…

Johnny: Ok then, this is a simple functionality, so let me suggest some user stories as I understand what you already said and you will correct me where I am wrong. Here we go:

1. **In order to** know that the calculator is turned on, **As a** tax payer **I want** to see "0" on the screen as soon as I turn it on.
2. **In order to** see what numbers I am currently operating on, **As a** tax payer, **I want** the calculator to display the values I enter
3. **In order to** calculate the sum of my different incomes, **As a** tax payer **I want** the calculator to enable addition of multiple numbers

What do you think?

Jane: The stories pretty much reflect what I want for the first iteration. I don't think I have any corrections to make.

Johnny: Now we'll take each story and collect some examples of how it should work.

Benjamin: Johnny, don't you think it is obvious enough to proceed with implementation straight away?

Johnny: Trust me, Benjamin, if there is one word I fear most in communication, it is "obvious". Miscommunication happens most often around things that people consider obvious, simply because other people do not.

Jane: Ok, I'm in. What do I do?

Johnny: Let's go through the stories one by one and see if we can find some key examples of how the features should work. The first story is…

In order to know that the calculator is turned on, As a tax payer I want to see “0” on the screen as soon as I turn it on.

Jane: I don’t think there’s much to talk about. If you display “0”, I will be happy. That’s all.

Johnny: Let’s write this example down using a table:

key sequence	Displayed output	Notes
N/A	0	Initial displayed value

Benjamin: That makes me wonder… what should happen when I press “0” again at this stage?

Johnny: Good catch, that’s what these examples are for – they make our thinking concrete. As Ken Pugh says[5]: “Often the complete understanding of a concept does not occur until someone tries to use the concept”. Normally, we would put the “pressing zero multiple times” example on a TODO list and leave it for later, because it’s a part of a different story. However, it looks like we’re done with the current story, so let’s move straight ahead. The next story is about displaying entered digits. How about it, Jane?

Jane: Agree.

Johnny: Benjamin?

Benjamin: Yes, go ahead.

In order to see what numbers I am currently operating on, As a tax payer, I want the calculator to display the values I enter

Johnny: Let’s begin with the case raised by Benjamin. What should happen when I input “0” multiple times after I only have “0” on the display?

Jane: A single “0” should be displayed, no matter how many times I press “0”.

Johnny: Do you mean this?

key sequence	Displayed output	Notes
0,0,0	0	Zero is a special case – it is displayed only once

Jane: That’s right. Other than this, the digits should just show on the screen, like this:

key sequence	Displayed output	Notes
1,2,3	123	Entered digits are displayed

Benjamin: How about this:

key sequence	Displayed output	Notes
1,2,3,4,5,6,7,1,2,3,4,5,6	1234567123456?	Entered digits are displayed?

[5]K. Pugh, Prefactoring, O’Reilly Media, 2005

Jane: Actually, no. My old calculator app has a limit of six digits that I can enter, so it should be:

key sequence	Displayed output	Notes
1,2,3,4,5,6,7,1,2,3,4,5,6	123456	Display limited to six digits

Johnny: Another good catch, Benjamin!

Benjamin: I think I'm beginning to understand why you like working with examples!

Johnny: Good. Is there anything else, Jane?

Jane: No, that's pretty much it. Let's start working on another story.

In order to calculate sum of my different incomes, As a tax payer I want the calculator to enable addition of multiple numbers

Johnny: Is the following scenario the only one we have to support?

key sequence	Displayed output	Notes
2,+,3,+,4,=	9	Simple addition of numbers

Jane: This scenario is correct, however, there is also a case when I start with "+" without inputting any number before. This should be treated as adding to zero:

key sequence	Displayed output	Notes
+,1,=	1	Addition shortcut – treated as 0+1

Benjamin: How about when the output is a number longer than six digits limit? Is it OK that we truncate it like this?

key sequence	Displayed output	Notes
9,9,9,9,9,9,+,9,9,9,9,9,9,=	199999	Our display is limited to six digits only

Jane: Sure, I don't mind. I don't add such big numbers anyway.

Johnny: There is still one question we missed. Let's say that I input a number, then press "+" and then another number without asking for result with "=". What should I see?

Jane: Every time you press "+", the calculator should consider entering current number finished and overwrite it as soon as you press any other digit:

key sequence	Displayed output	Notes
2,+,3	3	Digits entered after + operator are treated as digits of a new number, the previous one is stored

Jane: Oh, and just asking for result just after the calculator is turned on should result in "0".

key sequence	Displayed output	Notes
=	0	Result key in itself does nothing

Johnny: Let's sum up our discoveries:

key sequence	Displayed output	Notes
N/A	0	Initial displayed value
1,2,3	123	Entered digits are displayed
0,0,0	0	Zero is a special case – it is displayed only once
1,2,3,4,5,6,7	123456	Our display is limited to six digits only
2,+,3	3	Digits entered after + operator are treated as digits of a new number, the previous one is stored
=	0	Result key in itself does nothing
+,1,=	1	Addition shortcut – treated as 0+1
2,+,3,+,4,=	9	Simple addition of numbers
9,9,9,9,9,9,+,9,9,9,9,9,9,=	199999	Our display is limited to six digits only

Johnny: The limiting of digits displayed looks like a whole new feature, so I suggest we add it to the backlog and do it in another sprint. In this sprint, we will not handle such situation at all. How about that, Jane?

Jane: Fine with me. Looks like a lot of work. Nice that we discovered it up-front. For me, the limiting capability seemed so obvious that I didn't even think it would be worth mentioning.

Johnny: See? That's why I don't like the word "obvious". Jane, we will get back to you if any more questions arise. For now, I think we know enough to implement these three stories for you.

Jane: good luck!

Akt 3: Test-Driven Development

Benjamin: Wow, that was cool. Was that Acceptance Test-Driven Development?

Johnny: In a greatly simplified version, yes. The reason I took you with me was to show you the similarities between working with customer the way we did and working with the code using TDD process. They are both applying the same set of principles, just on different levels.

Benjamin: I'm dying to see it with my own eyes. Shall we start?

Johnny: Sure. If we followed the ATDD process, we would start writing what we call acceptance-level specification. In our case, however, a unit-level specification will be enough. Let's take the first example:

Statement 1: Calculator should display 0 on creation

key sequence	Displayed output	Notes
N/A	0	Initial displayed value

Johnny: Benjamin, try to write the first Statement.

Benjamin: Oh boy, I don't know how to start.

Johnny: Start by writing the statement in plain English. What should the calculator do?

Benjamin: It should display "0" when I turn the application on.

Johnny: In our case, "turning on" is creating a calculator. Let's write it down as a method name:

```
1  public class CalculatorSpecification
2  {
3
4  [Fact] public void
5  ShouldDisplay0WhenCreated()
6  {
7
8  }
9
10 }
```

Benjamin: Why is the name of the class `CalculatorSpecification` and the name of the method `ShouldDisplay0WhenCreated`?

Johnny: It is a naming convention. There are many others, but this is the one that I like. In this convention, the rule is that when you take the name of the class without the `Specification` part followed by the name of the method, it should form a legit sentence. For instance, if I apply it to what we wrote, it would make a sentence: "Calculator should display `0` when created".

Benjamin: Ah, I see now. So it's a statement of behavior, isn't it?

Johnny: That's right. Now, the second trick I can sell to you is that if you don't know what code to start your Statement with, start with the expected result. In our case, we are expecting that the behavior will end up as displaying "0", right? So let's just write it in the form of an assertion.

Benjamin: You mean something like this?

```
public class CalculatorSpecification
{

[Fact] public void
ShouldDisplay0WhenCreated()
{
 Assert.Equal("0", displayedResult);
}

}
```

Johnny: Precisely.

Benjamin: But that doesn't even compile. What use is it?

Johnny: The code not compiling is the feedback that you needed to proceed. While before you didn't know where to start, now you have a clear goal – make this code compile. Firstly, where do you get the displayed value from?

Benjamin: From the calculator display, of course!

Johnny: Then write down how you get the value from the display.

Benjamin: Like how?

Johnny: Like this:

```
public class CalculatorSpecification
{

[Fact] public void
ShouldDisplay0WhenCreated()
{
 var displayedResult = calculator.Display();

 Assert.Equal("0", displayedResult);
}

}
```

Benjamin: I see. Now the calculator is not created anywhere. I need to create it somewhere now or it will not compile - this is how I know that it's my next step. Is this how it works?

Johnny: Yes, you are catching on quickly.

Benjamin: Ok then, here goes:

```
public class CalculatorSpecification
{

[Fact] public void
ShouldDisplay0WhenCreated()
{
 var calculator = new Calculator();

 var displayedResult = calculator.Display();

 Assert.Equal("0", displayedResult);
}

}
```

Johnny: Bravo!

Benjamin: The code doesn't compile yet, because I don't have the `Calculator` class defined at all...

Johnny: Sounds like a good reason to create it.

Benjamin: OK.

```
public class Calculator
{
}
```

Benjamin: Looks like the `Display()` method is missing too. I'll add it.

```
public class Calculator
{
  public string Display()
  {
    return "0";
  }
}
```

Johnny: Hey hey, not so fast!

Benjamin: What?

Johnny: You already provided an implementation of `Display()` that will make our current Statement true. Remember its name? `ShouldDisplay0WhenCreated` – and that's exactly what the code you wrote does. Before we arrive at this point, let's make sure this Statement can ever be evaluated as false. You won't achieve this by providing a correct implementation out of the box. So for now, let's change it to this:

```
public class Calculator
{
  public string Display()
  {
    return "Once upon a time in Africa";
  }
}
```

Johnny: Look, now we can run the Specification and watch that Statement evaluate to false, because it expects "0", but gets "Once upon a time in Africa".

Benjamin: Running... Ok, it is false. By the way, do you always use such silly values to make Statements false?

Johnny: Hahaha, no, I just did it to emphasize the point. Normally, I would write `return "";` or something similarly simple. Now we can evaluate the Statement and see it turn false. Hence, we're sure that we have not yet implemented what is required for the Statement to be true.

Benjamin: I think I get it. For now, the Statement shows that we do not have something we need and gives us a reason to add this "thing". When we do so, this Statement will show that we do have what we need. So what do we do now?

Johnny: Write the simplest thing that makes this Statement true.

Benjamin: like this?

```
public class Calculator
{
  public string Display()
  {
    return "0";
  }
}
```

Johnny: Yes.

Benjamin: But that is not a real implementation. What is the value behind putting in a hardcoded string? The final implementation is not going to be like this for sure!

Johnny: You're right. The final implementation is most probably going to be different. What we did, however, is still valuable because:

1. You're one step closer to implementing the final solution
2. This feeling that this is not the final implementation points you towards writing more Statements. When there is enough Statements to make your implementation complete, it usually means that you have a complete Specification of class behaviors as well.

3. If you treat making every Statement true as an achievement, this practice allows you to evolve your code without losing what you already achieved. If by accident you break any of the behaviors you've already implemented, the Specification is going to tell you because one of the existing Statements that were previously true will turn false. You can then either fix it or undo your changes using version control and start over from the point where all existing Statements were true.

Benjamin: Ok, so it looks like there are some benefits after all. Still, I'll have to get used to this kind of working.

Johnny: Don't worry, this approach is an important part of TDD, so you will grasp it in no time. Now, before we go ahead with the next Statement, let's look at what we already achieved. First, we wrote a Statement that turned out false. Then, we wrote just enough code to make the Statement true. Time for a step called Refactoring. In this step, we will take a look at the Statement and the code and remove duplication. Can you see what is duplicated between the Statement and the code?

Benjamin: both of them contain the literal "0". The Statement has it here:

```
Assert.Equal("0", displayedResult);
```

and the implementation here:

```
return "0";
```

Johnny: Good, let's eliminate this duplication by introducing a constant called `InitialValue`. The Statement will now look like this:

```
[Fact] public void
ShouldDisplayInitialValueWhenCreated()
{
 var calculator = new Calculator();

 var displayedResult = calculator.Display();

 Assert.Equal(Calculator.InitialValue, displayedResult);
}
```

and the implementation:

```
public class Calculator
{
  public const string InitialValue = "0";
  public string Display()
  {
    return InitialValue;
  }
}
```

Benjamin: The code looks better and having the "0" constant in one place will make it more maintainable. However, I think the Statement in its current form is weaker than before. I mean, we can change the `InitialValue` to anything and the Statement will still be true, since it does not state that this constant needs to have a value of "0".

Johnny: That's right. We need to add it to our TODO list to handle this case. Can you write it down?

Benjamin: Sure. I will write it as "TODO: 0 should be used as an initial value."

Johnny: Ok. We should handle it now, especially since it's part of the story we are currently implementing, but I will leave it for later just to show you the power of TODO list in TDD – whatever is on the list, we can forget and get back to when we have nothing better to do. Our next item from the list is this:

Statement 2: Calculator should display entered digits

key sequence	Displayed output	Notes
1,2,3	123	Entered digits are displayed

Johnny: Benjamin, can you come up with a Statement for this behavior?

Benjamin: I'll try. Here goes:

```
[Fact] public void
ShouldDisplayEnteredDigits()
{
  var calculator = new Calculator();

  calculator.Enter(1);
  calculator.Enter(2);
  calculator.Enter(3);
  var displayedValue = calculator.Display();

  Assert.Equal("123", displayedValue);
}
```

Johnny: I see that you're learning fast. You got the parts about naming ans structuring a Statement right. There's one thing we will have to work on here though.

Benjamin: What is it?

Johnny: When we talked to Jane, we used examples with real values. These real values were extremely helpful in pinning down the corner cases and uncovering missing scenarios. They were easier to imagine as well, so they were a perfect suit for conversation. If we were automating these examples on acceptance level, we would use those real values as well. When we write unit-level Statements, however, we use a different technique to get this kind of specification more abstract. First of all, let me enumerate the weaknesses of the approach you just used:

1. Making a method `Enter()` accept an integer value suggests that one can enter more than one digit at once, e.g. `calculator.Enter(123)`, which is not what we want. We could detect such cases and throw exceptions if the value is outside the 0-9 range, but there are better ways when we know we will only be supporting ten digits (0,1,2,3,4,5,6,7,8,9).
2. The Statement does not clearly show the relationship between input and output. Of course, in this simple case it's pretty self-evident that the sum is a concatenation of entered digits. In general case, however, we don't want anyone reading our Specification in the future to have to guess such things.
3. The name of the Statement suggests that what you wrote is true for any value, while in reality, it's true only for digits other than "0", since the behavior for "0" is different (no matter how many times we enter "0", the result is just "0"). There are some good ways to communicate it.

Hence, I propose the following:

```
[Fact] public void
ShouldDisplayAllEnteredDigitsThatAreNotLeadingZeroes()
{
 //GIVEN
 var calculator = new Calculator();
 var nonZeroDigit = Any.Besides(DigitKeys.Zero);
 var anyDigit1 = Any.Of<DigitKeys>();
 var anyDigit2 = Any.Of<DigitKeys>();

 //WHEN
 calculator.Enter(nonZeroDigit);
 calculator.Enter(anyDigit1);
 calculator.Enter(anyDigit2);

 //THEN
 Assert.Equal(
  string.Format("{0}{1}{2}",
   (int)nonZeroDigit,
```

```
    (int)anyDigit1,
    (int)anyDigit2
   ),
   calculator.Display()
  );
}
```

Benjamin: Johnny, I'm lost! Can you explain what's going on here?

Johnny: Sure, what do you want to know?

Benjamin: For instance, what is this `DigitKeys` type doing here?

Johnny: It is supposed to be an enumeration (note that it does not exist yet, we just assume that we have it) to hold all the possible digits a user can enter, which are from the range of 0-9. This is to ensure that the user will not write `calculator.Enter(123)`. Instead of allowing our users to enter any number and then detecting errors, we are giving them a choice from among only the valid values.

Benjamin: Now I get it. So how about the `Any.Besides()` and `Any.Of()`? What do they do?

Johnny: They are methods from a small utility library I'm using when writing unit-level Specifications. `Any.Besides()` returns any value from enumeration besides the one passed as an argument. Hence, the call `Any.Besides(DigitKeys.Zero)` means "any of the values contained in DigitKeys enumeration, but not DigitKeys.Zero".

The `Any.Of()` is simpler – it just returns any value in an enumeration.

Note that by saying:

```
var nonZeroDigit = Any.Besides(DigitKeys.Zero);
var anyDigit1 = Any.Of<DigitKeys>();
var anyDigit2 = Any.Of<DigitKeys>();
```

I specify explicitly, that the first value entered must be other than "0" and that this constraint does not apply to the second digit, the third one and so on.

By the way, this technique of using generated values instead of literals has its own principles and constraints which you have to know to use it effectively. Let's leave this topic for now and I promise I'll give you a detailed lecture on it later. Agreed?

Benjamin: You better do, because for now, I feel a bit uneasy with generating the values – it seems like the Statement we are writing is getting less deterministic this way. The last question – what about those weird comments you put in the code? `GIVEN`? `WHEN`? `THEN`?

Johnny: Yes, this is a convention that I use, not only in writing, but in thinking as well. I like to think about every behavior in terms of three elements: assumptions (given), trigger (when) and expected result (then). Using the words, we can summarize the Statement we are writing in the following

way: "**Given** a calculator, **when** I enter some digits, the first one being non-zero, **then** they should all be displayed in the order they were entered". This is also something that I will tell you more about later.

Benjamin: Sure, for now I need just enough detail to be able to keep going – we can talk about the principles, pros and cons later. By the way, the following sequence of casts looks a little bit ugly:

```
string.Format("{0}{1}{2}",
 (int)nonZeroDigit,
 (int)anyDigit1,
 (int)anyDigit2
)
```

Johnny: We will get back to it and make it "smarter" in a second after we make this statement true. For now, we need something obvious. Something we know works. Let's evaluate this Statement. What is the result?

Benjamin: Failed: expected "351", but was "0".

Johnny: Good, now let's write some code to make this Statement true. First, we're going to introduce an enumeration of digits. This enum will contain the digit we use in the Statement (which is `DigitKeys.Zero`) and some bogus values:

```
public enum DigitKeys
{
 Zero = 0,
 TODO1, //TODO - bogus value for now
 TODO2, //TODO - bogus value for now
 TODO3, //TODO - bogus value for now
 TODO4, //TODO - bogus value for now
}
```

Benjamin: What's with all those bogus values? Shouldn't we correctly define values for all the digits we support?

Johnny: Nope, not yet. We still don't have a Statement which would say what digits are supported and which would make us add them, right?

Benjamin: You say you need a Statement for an element to be in an enum?

Johnny: This is a specification we are writing, remember? It should say somewhere which digits we support, shouldn't it?

Benjamin: It's difficult to agree with, I mean, I can see the values in the enum, should I really test for something when there's not complexity involved?

Johnny: Again, we're not only testing, we're specifying. I will try to give you more arguments later. For now, just bear with me and note that when we get to specify the enum elements, adding such Statement will be almost effortless.

Benjamin: OK.

Johnny: Now for the implementation. Just to remind you – what we have so far looks like this:

```
1  public class Calculator
2  {
3   public const string InitialValue = "0";
4   public string Display()
5   {
6    return InitialValue;
7   }
8  }
```

This clearly does not support displaying multiple digits (as we just proved, because the Statement saying they are supported turned out false). So let's change the code to handle this case:

```
1  public class Calculator
2  {
3   public const string InitialValue = "0";
4   private int _result = 0;
5
6   public void Enter(DigitKeys digit)
7   {
8    _result *= 10;
9    _result += (int)digit;
10  }
11
12  public string Display()
13  {
14   return _result.ToString();
15  }
16 }
```

Johnny: Now the Statement is true so we can go back to it and make it a little bit prettier. Let's take a second look at it:

```
[Fact] public void
ShouldDisplayAllEnteredDigitsThatAreNotLeadingZeroes()
{
 //GIVEN
 var calculator = new Calculator();
 var nonZeroDigit = Any.Besides(DigitKeys.Zero);
 var anyDigit1 = Any.Of<DigitKeys>();
 var anyDigit2 = Any.Of<DigitKeys>();

 //WHEN
 calculator.Enter(nonZeroDigit);
 calculator.Enter(anyDigit1);
 calculator.Enter(anyDigit2);

 //THEN
 Assert.Equal(
  string.Format("{0}{1}{2}",
   (int)nonZeroDigit,
   (int)anyDigit1,
   (int)anyDigit2
  ),
  calculator.Display()
 );
}
```

Johnny: Remember you said that you don't like the part where `string.Format()` is used?

Benjamin: Yeah, it seems a bit unreadable.

Johnny: Let's extract this part into a utility method and make it more general – we will need a way of constructing expected displayed output in many of our future Statements. Here is my go at this helper method:

```
string StringConsistingOf(params DigitKeys[] digits)
{
 var result = string.Empty;

 foreach(var digit in digits)
 {
  result += (int)digit;
 }
 return result;
}
```

Note that this is more general as it supports any number of parameters. And the Statement after this extraction looks like this:

```
[Fact] public void
ShouldDisplayAllEnteredDigitsThatAreNotLeadingZeroes()
{
 //GIVEN
 var calculator = new Calculator();
 var nonZeroDigit = Any.Besides(DigitKeys.Zero);
 var anyDigit1 = Any.Of<DigitKeys>();
 var anyDigit2 = Any.Of<DigitKeys>();

 //WHEN
 calculator.Enter(nonZeroDigit);
 calculator.Enter(anyDigit1);
 calculator.Enter(anyDigit2);

 //THEN
 Assert.Equal(
  StringConsistingOf(nonZeroDigit, anyDigit1, anyDigit2),
  calculator.Display()
 );
}
```

Benjamin: Looks better to me. The Statement is still evaluated as true, which means we got it right, didn't we?

Johnny: Not exactly. With moves such as this one, I like to be extra careful and double check whether the Statement still describes the behavior accurately. To make sure that's still the case, let's comment out the body of the `Enter()` method and see if this Statement would still turn out false:

```
public void Enter(DigitKeys digit)
{
 //_result *= 10;
 //_result += (int)digit;
}
```

Benjamin: Running... Ok, it is false now. Expected "243", got "0".

Johnny: Good, now we're pretty sure it works OK. Let's uncomment the lines we just commented out and move forward.

Benjamin: But wait, there is one thing that troubles me.

Johnny: I think I know - I was wondering if you'd catch it. Go ahead.

Benjamin: What troubles me is these two lines:

```
public const string InitialValue = "0";
private int _result = 0;
```

Isn't this a duplication? I mean, it's not exactly code duiplication, but in both lines, the value of `0` has the same intent. Shouldn't we remove this duplication somehow?

Johnny: Yes, let's do it. My preference would be to change the `InitialValue` to `int` instead of string and use that. But I can't do it in a single step as I have the two Statements depending on `InitialValue` being a string. if I just changed the type to `int`, I would break those tests asd well as the implementation and I always want to be fixing one thing at a time.

Benjamin: So what do we do?

Johnny: Well, my first step would be to go to the Statements that use `InitialValue` and use a `ToString()` method there. For example, in the Statement `ShouldDisplayInitialValueWhenCreated()`, I have an assertion:

```
Assert.Equal(Calculator.InitialValue, displayedResult);
```

which I can change to:

```
Assert.Equal(Calculator.InitialValue.ToString(), displayedResult);
```

Benjamin: But calling `ToString()` on a `string` just returns the same value, what's the point?

Johnny: The point is to make the type of whatever's on the left side of `.ToString()` irrelevant. Then I will be able to change that type without breaking the Statement. The new implementation of `Calculator` class will look like this:

```
public class Calculator
{
 public const int InitialValue = 0;
 private int _result = InitialValue;

 public void Enter(DigitKeys digit)
 {
  _result *= 10;
  _result += (int)digit;
 }

 public string Display()
 {
  return _result.ToString();
 }
}
```

Benjamin: Oh, I see. And the Statements are still evaluated as true.

Johnny: Yes. Shall we take on another Statement?

Statement 3: Calculator should display only one zero digit if it is the only entered digit even if it is entered multiple times

Johnny: Benjamin, this should be easy for you, so go ahead and try it. It is really a variation of the previous Statement.

Benjamin: Let me try... ok, here it is:

```
[Fact] public void
ShouldDisplayOnlyOneZeroDigitWhenItIsTheOnlyEnteredDigitEvenIfItIsEnteredMultipleTim\
es()
{
 //GIVEN
 var calculator = new Calculator();

 //WHEN
 calculator.Enter(DigitKeys.Zero);
 calculator.Enter(DigitKeys.Zero);
 calculator.Enter(DigitKeys.Zero);

 //THEN
 Assert.Equal(
  StringConsistingOf(DigitKeys.Zero),
  calculator.Display()
 );
}
```

Johnny: Good, you're learning fast! Let's evaluate this Statement.

Benjamin: It seems that our current code already fulfills the Statement. Should I try to comment some code to make sure this Statement can fail just like you did in the previous Statement?

Johnny: That would be a wise thing to do. When a Statement turns out true without requiring you to change any production code, it's always suspicious. Just like you said, we have to change production code for a second to force this Statement to become false, then undo this modification to make it true again. This isn't as obvious as previously, so let me do it. I will mark all the added lines with `//+` comment so that you can see them easily:

```
public class Calculator
{
 public const int InitialValue = 0;
 private int _result = InitialValue;
 string _fakeResult = "0"; //+

 public void Enter(DigitKeys digit)
 {
  _result *= 10;
  _result += (int)digit;
  if(digit == DigitKeys.Zero) //+
  {  //+
   _fakeResult += "0";  //+
  } //+
 }

 public string Display()
 {
  if(_result == 0)  //+
  {  //+
   return _fakeResult;  //+
  }  //+
  return _result.ToString();
 }
}
```

Benjamin: Wow, looks like a lot of code just to make the Statement false! Is it worth the hassle? We will undo this whole change in a second anyway…

Johnny: Depends on how confident you want to feel. I would say that it's usually worth it – at least you know that you got everything right. It might seem like a lot of work, but it only took me about a minute to add this code and imagine you got it wrong and had to debug it on a production environment. Now *that* would be a waste of time.

Benjamin: Ok, I think I get it. Since we saw this Statement turn false, I will undo this change to make it true again.

Johnny: Sure.

Epilog

Czas zostawić Johnny'ego i Benjamina, przynajmniej na razie. Tak naprawdę planowałem by ten rozdział był dłuższy i objął wszystkie ważne działania, ale obawiam się, że już jest zbyt długi i Cię

nudzę. Powinieneś teraz kojarzyć, jak wygląda cykl TDD, zwłaszcza, że Johnny i Benjamin poruszyli w międzyczasie wiele wątków. Powrócę do nich w dalszej części książki. Jeśli czujesz się zagubiony lub nieprzekonany w którymkolwiek z tematów poruszonym przez Johnny'ego, nie martw się – nie oczekuję, że będziesz od razu biegły w zakresie technik przedstawionych w tym rozdziale. Przyjdzie na to czas.

Odnajdźmy się odrobinę

W ostatnim rozdziale nastąpiła ożywiona rozmowa między Johnnym i Benjaminem. Nawet podczas tak krótkiej sesji Benjamin, jako nowicjusz TDD, miał wiele pytań i wiele rzeczy potrzebował jeszcze ustalić. Zbierzmy wszystkie pytania, na które jeszcze nie udzielono odpowiedzi i spróbujemy odpowiedzieć na nie w następnych rozdziałach. Oto pytania:

- Jak nazwać wymaganie?
- Jak rozpocząć pisanie wymagania?
- Co mówi TDD o analizie wymagań i co, w zasadzie, znaczy "GIVEN-WHEN-THEN"?
- Jaki dokładnie jest zakres wymagania? Klasa, metoda lub coś innego?
- Jaka jest rola listy TODO w TDD?
- Dlaczego warto używać generowanych, anonimizowanych wartości zamiast literałów jako danych wejściowych testowanego zachowania?
- Po co i w jaki sposób korzystać z klasy `Any`?
- Jaki kod wyodrębnić z wymagania do stworzenia pomocniczej, współdzielonej biblioteki?
- Skąd takie dziwne podejście do tworzenia stałych - przy pomocy typu wyliczeniowego enum?

Wiele pytań, prawda? Niefortunnie, TDD ma wysoki próg wejścia, przynajmniej dla kogoś przyzwyczajonego do tradycyjnego sposobu pisania kodu. W każdym razie, ten samouczek służy do znalezienia odpowiedzi na takie pytania i obniżeniu tego progu. Tak więc, postaramy się odpowiedzieć na te pytania jedno po drugim.